FRANZIS KOMPAKT

Klein
68000 kompakt

Der Assembler-Befehlssatz der 680xx-Prozessoren



FRANZIS KOMPAKT

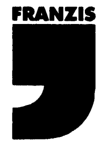
FRANZIS KOMPAKT

Rolf-Dieter Klein

68000 kompakt

Der Assembler-Befehlssatz der 680xx-Prozessoren

Mit 10 Abbildungen



CIP-Titelaufnahme der Deutschen Bibliothek

Klein, Rolf-Dieter:
68000 kompakt : der Assembler-Befehlssatz der 680xx-Prozessoren / Rolf-Dieter Klein. – München : Franzis, 1989
(Franzis kompakt)
ISBN 3-7723-7632-0

Umschlaggestaltung: Kaselow Design, München

Satz: Dieter Kriesell, Lonnerstadt
Druck: Kösel, Kempten
Printed in Germany · Imprimé en Allemagne

ISBN 3-7723-7632-0

Vorwort

Der vorliegende Band gehört zur Reihe FRANZIS KOMPAKT. Mit dieser Reihe will der Verlag Ihnen Nachschlagewerke in die Hand geben, die alle wesentlichen Informationen in knappster Form darbieten, so daß sie bei der Arbeit am Computer benutzt werden können.

Dieser Band ist so gestaltet, daß er Ihnen als schnelle Nachschlagehilfe beim Assemblerprogrammieren der 68000er Familie dienen kann. Dabei sind die Befehle der Prozessoren 68008, 68000, 68010 und 68012 berücksichtigt.

Inhalt

1 Einleitung

In diesem Band sind die Befehle der Prozessoren 68008, 68000, 68010 und 68012 beschrieben.
Im ersten Teil (Kapitel 2) werden die Hardwareeigenschaften der Prozessoren beschrieben. Damit erhalten Sie einen Überblick und können die Prozessoren voneinander unterscheiden.
Im zweiten Teil (Kapitel 3) wird die für Programmierer relevante Struktur beschrieben, so die Registerstruktur, Adressierarten usw.
Im dritten Teil (Kapitel 4), der den größten Teil des Bandes einnimmt, sind die einzelnen Befehle aufgeführt. Neben der Bedeutung des Befehls erfahren Sie dort Informationen über die Assemblersyntax, Eigenschaften des Befehls, wie Änderung der Flag-Register, und die genaue Befehlsbeschreibung. Ein Programmbeispiel zeigt die Befehlsanwendung. Als Assembler wurde in diesem Band die Syntax des NDR-Assemblers verwendet, die sich aber meistens mit anderen Assemblern von Rechnern wie Amiga, Atari oder Macintosh deckt, wenn diese sich an die von Motorola definierte Syntax halten.

2 Die Mikroprozessorfamilie 680xx

Alle 680xx Prozessoren haben einen ähnlichen internen Aufbau. Die 680xx gehören mit Ausnahme des 68020 zu den 16-Bit Prozessoren, wobei sie alle Merkmale eines 32-Bit-Prozessors besitzen. Der 68020 ist ein echter 32-Bit Prozessor.
Der 68008 besitzt als einziger Prozessor dieser Familie einen 8-Bit-Datenbus, womit sich sehr kostengünstig kleine Systeme aufbauen lassen. *Abb. 2.1* zeigt einen Vergleich. Der 68000, 68010 und 68012 besitzen einen 16-Bit-Datenbus, RAM und ROM müssen über 16-Bit-Datenbusse angesprochen werden. Peripheriebausteine können auch über einen 8-Bit-Datenbus angesprochen werden. Der 68020 besitzt einen dynamisch rekonfigurierbaren Datenbus, der sowohl 32-Bit, 16-Bit oder 8-Bit breit sein kann, und sowohl Speicher als auch Peripherie lassen sich damit betreiben.
Die 680xx - Prozessoren verwenden alle ein sogenanntes "memory map" als Adressierung für Peripherie-Bausteine. Das bedeutet, ein Teil des Gesamtadreßraums für den Speicher wird zur Adressierung der Peripheriebausteine verwendet. Der Vorteil dieses Verfahrens liegt darin, daß alle Befehle, die für den Speicher verwendet werden können, auch bei Peripheriebausteinen angewandt werden dürfen.
Die Prozessoren 68008 und 68000 waren die ersten Bausteine, die auf den Markt kamen.
Sie bilden gewissermaßen die Basis der 680xx-Familie. Die Befehlssätze dieser beiden Prozessoren sind absolut identisch. Der 68008 besitzt auf seinem Chip gegenüber dem 68000 nur eine Zusatzlogik, die alle 16-Bit-Zugriffe in zwei 8-Bit-Zugriffe zerlegt.
Bei den Prozessoren 68010 und 68012 wurde eine wesentliche Erweiterung hinzugefügt, die virtuelle Speicherverwaltungsmöglichkeit. Dabei wird zusätzlich eine MMU verwendet, die bei einem Zugriff auf nicht vorhandenen Speicher den Buszyklus abbricht.

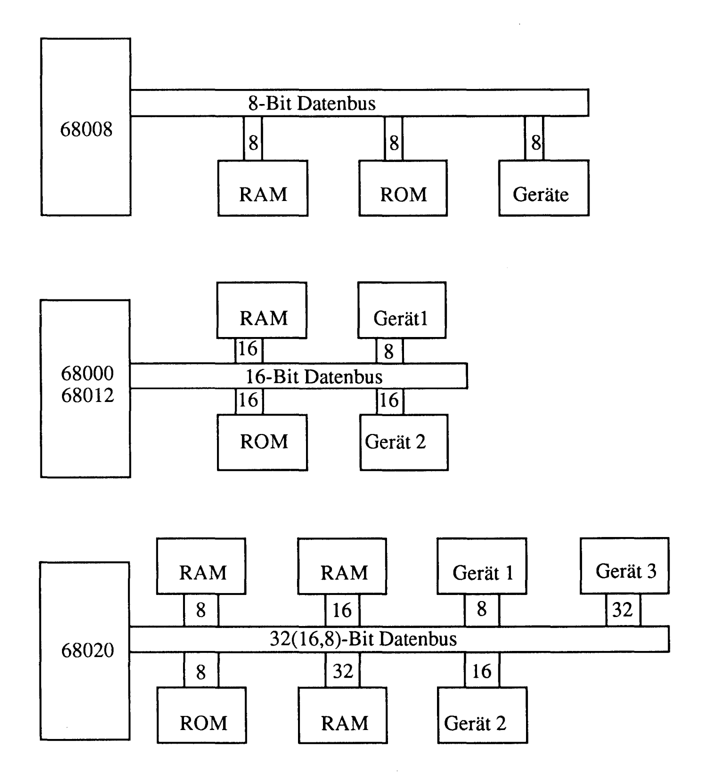


Abb. 2.1 Vergleich der Systeme

Danach wird durch ein Interruptprogramm der Speicher mit Daten von einem Hintergrundmedium (z.B. Platte) mit den fehlenden Daten gefüllt und dann wird der Buszyklus fortgeführt. Die MMU enthält Informationen, welcher logisch angesprochene Speicherbereich tatsächlich im Speicherbereich liegt, und wenn ja wo. Dabei wird auch eine Adreßumrechnung von logischen auf physikalische Adressen durchgeführt. Die Prozessoren 68010 und 68012 besitzen ferner ein paar zusätzliche Befehle, die in der Befehlsliste in diesem Band besonders gekennzeichnet sind.
Beim 68020 schließlich kam die Erweiterung auf 32-Bit. Daten- und Adreßleitungen sind mit 32 Bit herausgeführt. Ferner sind eine Vielzahl neuer Befehle und Adressierarten verfügbar, so z.B. Multiplikation 32Bit x 32Bit -> 64Bit und mehrfach indirekte Adressierung.

3 Der Befehlssatz der 680xx-Familie

3.1 Die Register

Abb. 3.1.1 zeigt eine Übersicht der vorhandenen Register.
Vorhanden sind die Register D0 bis D7, auch Datenregister genannt, dann die Adreßregister A0 bis A6, sowie zweimal das Adreßregister A7. Zusätzlich gibt es noch den Programmzähler sowie ein Statusregister. Alle Register mit Ausnahme des Statusregisters sind 32 Bit breit, das Statusregister besitzt nur 16 Bit. Beim 68010 und 68012 gibt es noch das Vektor-Basis-Register, sowie zwei 3-Bit breite Register, die einen Funktions-Code beinhalten können.

D0 bis D7:

Jedes dieser Datenregister ist 32 Bit breit. Die Register können aber auch 16- und 8-Bit Werte verwenden. Für ein Langwort werden alle 32 Bit verwendet, bei einem Wort sind es nur die Bitpositionen 0..15 und bei einem Byte werden die Stellen 0..7 verwendet.

A0 bis A7:

Sollen Speicherzellen oder Peripheriegeräte z.B. indirekt adressiert werden, so kommen die Adreßregister zum Einsatz. Diese Register sind nicht so universell wie die Datenregister, so können z.B. nicht alle Arithmetikbefehle ausgeführt werden. Andererseits gibt es besondere Befehle, die nur mit Adreßregistern arbeiten.
Das Adreßregister A7 besitzt einen Sonderstatus. Zunächst einmal gibt es zwei davon. Jedoch ist nur eines davon aktiv. Welches, das hängt von der Betriebsart des Prozessors ab. Befindet sich der

Das Registermodell der µP 68008/68010/68012:

31 16 15 8 7 0

D0

D1

D2

D3

D4

D5

D6

D7

31 16 15 0

A0

A1

A2

A3

A4

A5

A6

A7/USP

A7'/SSP

31 0

PC

15 8 7 0

CCR SR

31 0

VBR*

3 0

SFC*

DFC*

* nur 68010/68012:

VBR: Vektor-Basis-Register

SFC: Source Function Code Register

DFC: Destination Function Code Register

Abb. 3.1.1 Registerübersicht

Prozessor im sogenannten System-Mode, dann wird der SSP (System Stack Pointer) verwendet, und im User-Mode wird der USP (User Stack Pointer) aktiviert. Damit ist auch die Verwendung des Registers A7 angedeutet. Es wird als sogenannter Stackpointer verwendet. Unterprogramm-Rückkehradressen werden zum Beispiel mit Hilfe dieses Registers im Speicher abgelegt. Dazu enthält A7 die Adresse eines freien Speicherbereichs. Datenwerte werden mit absteigenden Adressen abgelegt , weshalb der Stackpointer zunächst die höchste verfügbare Adresse enthalten muß.

Programmzähler:

Der Programmzähler bestimmt, welcher Befehl als nächstes ausgeführt werden soll.
Er ist ebenfalls 32 Bit breit. Bei dem Prozessor 68008 werden nur 20 Bit davon verwendet, da er nicht mehr als 1 MByte adressieren kann, beim 68000 und 68010 sind es 24 Bit und beim 68020 werden alle 32 Bit verwendet. Daher sollte man davon Abstand nehmen, ungebrauchte Bits mit Zusatzinformationen zu belegen. Dies gilt natürlich auch für die Adreßregister.

Status-Register:

Abb. 3.1.2 zeigt die Belegung des Status-Registers. Es wird im Befehlssatz oft mit SR abgekürzt. Im Status-Register sind wichtige Zusatzinformationen untergebracht. So gibt es beim 68xxx zwei verschiedene Betriebsarten: den System-Mode und den User-Mode. In welchem Mode sich der Prozessor gerade befindet ist zum Beispiel in Bit 13 des Status-Registers vermerkt. Das Status-Register ist in zwei Hälften unterteilt, dem System-Teil und dem User-Teil. Der System-Teil kann nur vom System-Mode aus verändert werden. Bei den Prozessoren 68010 und 68020 kann der System-Teil auch nur im System-Mode gelesen werden. Im User-Teil sind allgemeine Status-

Das Statusregister (SR) des µP 68000

System-Byte | Anwender-Byte (CCR)

T		S			I2	I1	I0				X	N	Z	V	C
15		13			10	9	8				4	3	2	1	0

Erläuterung:

T	Tracebit, gesetzt im Trace-Modus
S	Supervisorbit, gesetzt im Supervisor-Modus
I2/I1/I0	Interruptmaske
X	e**X**tend-Flag: Erweiterung
N	**N**egativ-Flag
Z	**Z**ero-Flag
V	o**V**erflow-Flag: Überlauf
C	**C**arry-Flag: Übertrag
(schraffiert)	nicht verwendet

Abb. 3.1.2 Status-Register

Informationen untergebracht, z.B. arithmetischer Überlauf, Vorzeichen usw.

Die Belegung des Status-Registers im einzelnen:

Bit 15 Trace-Mode aktiv, wenn Bit 15=1. Es wird genau ein Befehl ausgeführt und danach die TRACE-Exception ausgelöst. Um den Trace-Mode zu aktivieren, setzt man das Statusbit 15 z.B. im Statuswort auf dem Stack und führt einen RTE aus. Danach wird der Trace-Mode aktiviert und genau ein Befehl ausgeführt.
Bei der Trace-Exception wird der Trace-Mode vom Prozessor wieder ausgeschaltet. Auf diese Weise kann ein Programm im Einzelschritt durchlaufen werden.

Bit 13 System-Mode (auch Supervisor-Mode genannt), wenn Bit13=1.

Bit 10..8 Interrupt-Ebene. Werden alle drei Bits auf 1 gesetzt, so sind alle maskierbaren Interrupts gesperrt.

— CCR (Conditioncode-Register, User-Teil):

Bit 4 X-Flag, Extend, Erweiterung. Wird meist wie das C-Flag behandelt, bleibt jedoch bei manchen Operationen im alten Zustand, wenngleich
das C-Flag verändert wird.

Bit 3 N-Flag, Negative,Vorzeichen. Ist ein Ergebnis kleiner Null, so wird das Bit auf 1 gesetzt.

Bit 2 Z-Flag, Zero, Null. Ist ein Ergebnis gleich Null, so wird das Bit auf 1 gesetzt.

Bit 1 V-Flag, Overflow, Überlauf. Wird im allgemeinen bei einem arithmetischen Überlauf gesetzt, also wenn der Zahlenbereich durch die ausgeführte
Operation verlassen wurde.

Bit 0 C-Flag, Carry, Übertrag. Bei einem Übertrag in die nächste Stelle wird dieses Bit gesetzt.

Nicht alle Bits werden bei allen Befehlen belegt. Bei der Erläuterung der Befehle sind die jeweils gültigen Bedeutungen vermerkt.
Das CCR kann zum Beispiel auch durch Sprungbefehle abgefragt werden. Beim 68010 und 68012 sind die Bits 15 bis 8 des Status-Registers nur im System-Mode abfragbar.

Vektor-Basis-Register:

Beim 68010 und 68012 ist es möglich, die Anfangsadresse für die Exception-Vektoren in diesem Register festzulegen.

Funktions-Code-Register:

Da die 680xx-Prozessoren über die Hardwareanschlüsse FC0..FC2 Informationen über die aktuelle Betriebsart (System-Mode, User-Mode usw.) an die Außenwelt geben, ist es möglich z.B. mit einer MMU den Adreßbereich z.B. in User-RAM und System-RAM aufzuteilen. Ferner kann zwischen Programm und Datenbereich unterschieden werden. Bei den Prozessoren 68010 und 68012 ist es möglich vom System-Mode aus auf alle unterschiedlichen Adreßräume zuzugreifen. Dazu dienen die beiden Register SFC und DFC, die den jeweiligen Funktionscode für die Quelle und das Ziel einstellen.
Funktions-Codes:

FC2	FC1	FC0	
0	0	0	undefiniert
0	0	1	User-Mode Daten
0	1	0	User-Mode Programm
0	1	1	undefiniert
1	0	0	undefiniert
1	0	1	System-Mode Daten
1	1	0	System-Mode Programm
1	1	1	CPU-Space (Interrupt-Vektoren etc.)

3.2 Der Interrupt- und Trap-Mechanismus

Bei der 68xxx-Familie gibt es eine Vielzahl von Möglichkeiten der Programmunterbrechung, die bei der 68xxx-Familie Exceptions genannt wird. Übersetzt bedeutet das Wort Exception soviel wie Ausnahmebehandlung. Durch eine Exception wird die normale Programmausführung unterbrochen und ein jeweils spezielles Programm angesprungen. Es wird ferner zwischen externen und internen Exceptions unterschieden. Externe Exceptions werden durch Anlegen eines Signals an den Interrupt-Leitungen ausgelöst. Damit ist es der CPU möglich auf externe Ereignisse schnell zu reagieren. Interne Exceptions können z.B. durch eine Division durch Null ausgelöst werden oder durch einen speziellen Befehl, der TRAP genannt wird. Da jede Exception den System-Mode aktiviert, ist es so z.B. für Anwenderprogramme möglich, durch einen TRAP-Aufruf spezielle Betriebsystemdienste anzufordern. Alle Exceptions werden mit Hilfe einer Adreßtabelle an die entsprechenden Programmteile verteilt. Die Adreßtabelle liegt im Hauptspeicherbereich 0 bis $3ff. Bei den Prozessoren 68010 und 68020 ist die Lage der Tabelle programmierbar.
Die Adreßeinträge werden grundsätzlich mit Langworten durchgeführt.

Vektor 0, Adresse 0:

SSP-Wert (System-Stack A7), der nach einem Hardware-Reset geladen wird.

Vektor 1, Adresse 4:

Programmzählerwert, der nach einem Hardware-Reset geladen wird.

Beispiel:

```
    0000: DC.L $1FFFF        * Stack Start
    0004: DC.L PROGANFANG    * Start des Hauptprogramms
          ...                * ggf. weitere Vektoren
    0400:                    * erste freie Adresse
          ...                * Programm kann beliebig
      liegen
PROGANFANG: ...              * Programmstart
```

Vektor 2, Adresse 8:

Bus-Error: Diese Exception kann durch eine Hardwareleitung an der CPU ausgelöst werden, z.B. wenn eine Adresse angesprochen wird, auf der kein Speicher oder Peripheriebaustein liegt.

Vektor 3, Adresse $C:

Address-Error: Wird beim 68008, 68000 oder 68010 mit einem Wort oder Langwort auf eine ungerade Adresse zugegriffen, so wird diese Exception ausgelöst. Achtung: Beim 68020 erfolgt keine Exception, da es dort erlaubt ist mit allen Datentypen auch auf ungerade Adressen zuzugreifen. Achtung, auch Befehlscodes dürfen nicht auf ungeraden Adressen liegen.

Beispiel:

```
    LEA $1001,A0       * ungerade Adresse laden
    MOVE.W (A0),D0     * Wort laden
    —— Exception wird beim 68008,68000 und 68010 ausge-
löst.
```

Vektor 4, Adresse $10:

Illegal-Instruction: Wird ein unbekannter Befehlscode ausgeführt, so erfolgt diese Exception. Mit dem speziellen Befehl ILLEGAL kann diese Exception ebenfalls ausgeführt werden.

Vektor 5, Adresse $14:

Zero-Divide: Die Exception erfolgt bei einer Division durch 0.

Beispiel:

```
MOVE #0,D0
MOVE.L #10,D1
DIVS D0,D1          * D1 / 0 ergibt Fehler.
— Exception wird ausgelöst.
```

Vektor 6, Adresse $18:

CHK-Instruction: Der Vergleichswert für CHK lag außerhalb der angegebenen Grenzen. Mit dem CHK-Befehl lassen sich z.B. Indexgrenzen überprüfen.
Beispiel:

```
MOVE #10,D0         * Zu prüfenende Zahl in D0
CHK #9,D0           * erlaubt 0..9
— Exception wird ausgelöst.
```

Vektor 7, Adresse $1C:

TRAPV-Instruction: Beim Aufruf von TRAPV war das Überlauf-Bit (V-Flag) gesetzt.

Beispiel:

```
MOVE.W #$7FFF,D0        * +32767
ADD.W #1,D0             * ergibt Überlauf
TRAPV                   * prüfen
— Exception wird ausgelöst.
```

Vektor 8, Adresse $20:

Privilege-Violation: Ein Befehl wird im User-Mode ausgeführt, obwohl nur im System-Mode zugelassen.

Beispiel:

```
... im User-Mode ...    * nur wenn im User-Mode
EORI #$8000,SR          * versucht SR zu ändern
— Exception
```

Vektor 9, Adresse $24:

Trace: Das Trace-Bit war im SR bei der Ausführung des letzten Befehls gesetzt. Damit läßt sich eine Einzelschritt-Funktion realisieren.

Vektor 10, Adresse $28:

Line 1010 Emulator: Alle Befehlscodes, die mit 1010 (oder sedezimal A) beginnen, lösen diese Exception aus. Damit ist es möglich z.B. eigene Befehle zu realisieren. Beispiel:

```
DC.W $A000              * Axxx genügt
— Exception wird ausgelöst
```

Vektor 11, Adresse $2C:

Line 1111 Emulator: Alle Befehlscodes, die mit 1111 (oder sedezimal F) beginnen, lösen diese Exception aus. Damit ist es möglich z.B. eigene Befehle zu realisieren.

Beispiel:

```
DC.W $F000          * Fxxx genügt
— Exception wird ausgelöst
```

Achtung: Beim 68020 werden zur Ansteuerung des Co-Prozessors 68881 (Arithmetik-Prozessor) einige dieser Line-F-Befehle verwendet. Daher sollte man sie bei eigenen Programmen nur zur Emulation des Co-Prozessors verwenden und nicht für eigene Zwecke.

Vektoren 12,13, Adresse $30,$34:

Sind für Erweiterungen reserviert.

Vektor 14, Adresse $38 (nur bei 68010 und 68012):

Format Error: Bei den Prozessoren 68010 und 68012 liegt ein geändertes Datenformat bei Exceptions auf dem Stack. Wenn das Datenformat bei einem RTE-Befehl nicht gültig ist, so wird diese Exception ausgelöst.

Vektor 15, Adresse $3C:

Uninitialized Interrupt Vektor: Wenn ein 68xxx-Peripheriebaustein einen Interrupt auslöst, aber der Interruptvektor in diesem Baustein noch nicht programmiert war, so wird dieser Vektor verwendet.

Vektoren 16-23, Adresse $40-$5C:

Nicht belegt, für Erweiterungen reserviert.

Vektor 24, Adresse $60:

Spurious Interrupt: Wenn während einer Interrupt-Bestätigung ein Bus-Error ausgelöst wird, so wird dieser Vektor verwendet.

Vektoren 25-31, Adressen $64-$7C:

Interrupt-Autovektor: Wird das Signal VPA bei einer Interrupt-Bestätigung angelegt, so erfolgt die Vektorauswahl anhand des Codes, der durch die Kombination von IPL2, IPL1 und IPL0 festgelegt ist. Werden alle drei Leitungen IPL2-IPL0 gleichzeitig auf Low gelegt, so wird zum Beispiel ein Interrupt Level 7 ausgelöst, der zudem nicht maskierbar ist. Der Vektor 31 wird dann verwendet. Achtung: beim 68008 sind IPL0 und IPL2 auf ein gemeinsames Pin geführt und fest miteinander verdrahtet.

Kombinationen:

IPL2	IPL1	IPL0	* negative Logik !
1	1	1	* kein Interrupt
1	1	0	* Interrupt Level 1, Adresse $64
1	0	1	* Interrupt Level 2, Adresse $68
1	0	0	* Interrupt Level 3, Adresse $6C
0	1	1	* Interrupt Level 4, Adresse $70
0	1	0	* Interrupt Level 5, Adresse $74
0	0	1	* Interrupt Level 6, Adresse $78
0	0	0	* Interrupt Level 7, Adresse $7C

Vektoren 32-47, Adressen $80-$BC:

Trap-Vektoren: Der Befehl TRAP #n verwendet einen dieser Vektoren. Dabei errechnet sich die Vektornummer (nr) wie folgt: nr = n + 32. Für n kann der Bereich 0..15 eingesetzt werden.

Beispiel:

    TRAP #1

Der Vektor 33 (nr = 1 + 32) wird verwendet und damit das Langwort auf Adresse $84 als Exception-Unterprogramm-Adresse interpretiert.

Vektoren 64-255, Adresse $100-$3FC:

Interrupt-Vektoren: Wenn ein Peripheriebaustein einen Vektor zur Verfügung stellt, kann er in diesen Bereich führen. Die 68xxx-Peripheriebausteine haben für diesen Zweck ein Vektorregister.

Exception-Stack-Anordnung:

für 68008 und 68000:

Bei einer Exception wird der Programmzählerinhalt (zeigt auf den nächsten Befehl) und der Inhalt des Statusregisters auf den Stack gerettet.
Der Inhalt des Stack sieht also so aus:

```
SSP-> Status-Register
      Programm-Zähler MSB
      Programm-Zähler LSB
```

Der Stackpointer zeigt dabei auf ein Datenwort in dem der Statusregister-Inhalt abgelegt ist.

für 68010 und 68012:

Es werden Zusatzinformationen auf dem Stack abgelegt:

SSP-> Status-Register
Programm-Zähler MSB
Programm-Zähler LSB
Format(4Bit) Vektor Offset
ggf. weitere Informationen
...

Der Format-Code bestimmt dabei die Anzahl der restlichen Worte, die sich auf dem Stack befinden:

Format		
	0000	keine weiteren Informationen, insgesamt 4 Worte
	1000	25 weitere Worte mit internen Daten, insgesamt 29 Worte
	Rest	reserviert (z.B. beim 68020 verwendet)

3.3 Die Adressierarten

Die 68000er Familie unterscheidet drei verschiedene Wortbreiten. Die kleinste Einheit ist ein Byte, also 8 Bit. Werden zwei Byte zusammengefaßt, so erhält man ein Wort, also 16 Bit. Zwei Worte lassen sich wiederum zu einem Langwort zusammenfassen, dann mit 32 Bit. Jeder Speicherplatz ist numeriert, dabei wird der Speicher immer in Bytes aufgeteilt, unabhängig davon, ob man den Speicher in Bytes (68008), Worte (68000) oder in Langworten (68020) organisiert hat. Adresse 0 spricht das erste Byte im Speicher an. Liest man ein Wort beginnend von Adresse 0, so gelangt das erste Byte auf die Bitpositionen 15 bis 8. Wenn ein Langwort von Adresse 0 gelesen wird, so gelangt das erste Byte auf die Bitpositionen 31 bis 24. Worte und Langworte dürfen bei den Prozessoren 68008, 68000 und 68010 nur auf geraden Adressen angesprochen werden, also 0, 2, 4, 6, 8, 10 usw. Wird das nicht befolgt, so wird eine Ausnahmebehandung ausgelöst und die Programmausführung unterbrochen. Der Prozessor 68020 hat diese Einschränkung nicht mehr.

Abb. 3.3.1 zeigt die Aufteilung im Speicher bei einer Byteadressierung. Die Abkürzung MSB steht dabei für Most Significant Byte, also höchstwertigste Stelle, und LSB für Least Significant Byte, also niederwertigste Stelle.

In *Abb 3.3.2* ist die Speicheraufteilung bei der Wortadressierung dargestellt und *Abb. 3.3.3* zeigt die Aufteilung bei einer Langwortadressierung. Beispielsweise soll das Langwort $AB1234CD ($ kennzeichnet sedezimale Zahlen) auf Adresse $1000 abgelegt werden. Der Befehl dazu lautet MOVE.L #$AB1234CD,$1000. Der Speicher ist dann wie folgt belegt, wenn er wortweise ausgelesen wird:

```
1000:   $AB12
1002:   $34CD
```

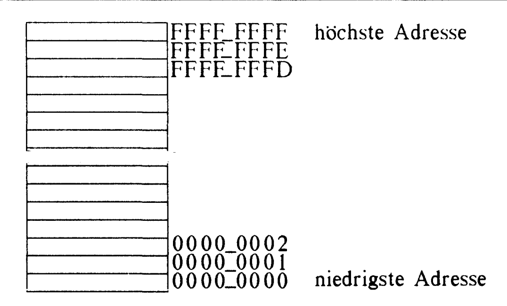


Abb. 3.3.1 Byteadressierung

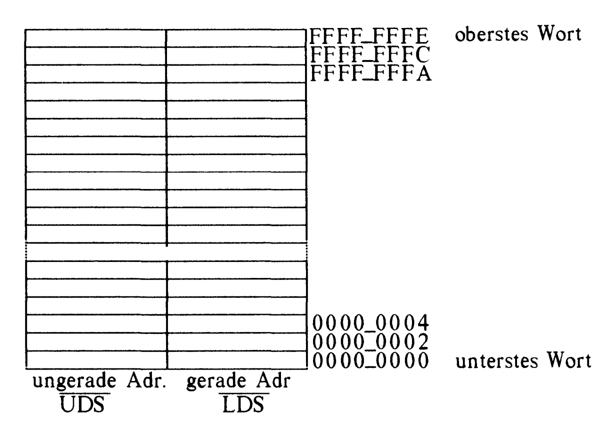


Abb. 3.3.2 Wortadressierung

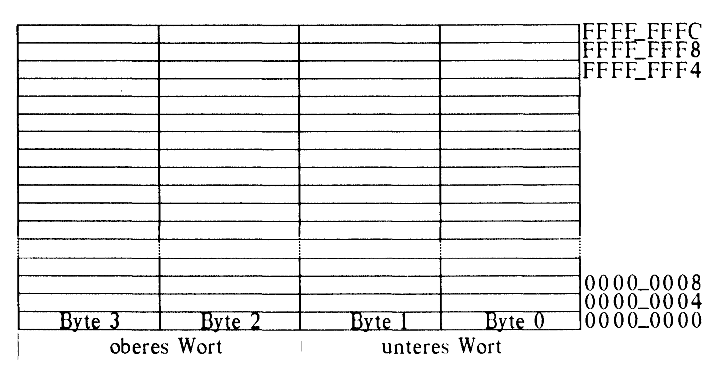


Abb. 3.3.3 Langwortadressierung

oder wenn der Speicher byteweise ausgelesen wird:

```
1000:   $AB
1001:   $12
1002:   $34
1003:   $CD
```

Um dem Übersetzungsprogramm (Assembler) zu sagen, welche Wortlänge gewünscht wird, gibt es Abkürzungen, die an den jeweiligen Befehl angehängt wird. ".B" steht für die Byteadressierung, ".W" für die Wortadressierung und ".L" wird bei einem Langwort verwendet. Wird keine Erweiterung angegeben, so wird im allgemeinen die Wortadressierung verwendet.

Beispiel:

```
MOVE.B #$12,$1000    * Wert $12 als Byte ablegen
MOVE.W #$12,$1000    * Wert $12  als Wort ablegen
MOVE.L #$12,$1000    * Wert $12  als Langwort ablegen
```

Im Speicher stehen nacheinander folgende Werte:

```
1000: xx xx xx xx    * Alle Speicherzellen undefiniert
1000: 12 xx xx xx    * nach MOVE.B
1000: 00 12 xx xx    * nach MOVE.W
1000: 00 00 00 12    * nach MOVE.L
```

Dn

Direkte Datenregister-Adressierung: Anstelle von "n" kann ein Zahl zwischen 0 und 7 stehen, die das Register bezeichnet. Jedes der Datenregister D0 bis D7 ist 32 Bit breit. Das Register wird durch diese

Adressierart direkt angesprochen. Beispiel:

```
MOVE.L D0,D1
```

Der Inhalt des Registers D0 wird in das Register D1 kopiert. Die Datenregister lassen sich sowohl als Langwort (".L"), aber auch als Wort (".W") oder Byte (".B") ansprechen.

An

Direkte Adreßregister-Adressierung: Anstelle von "n" steht eine Zahl im Bereich 0..7, die das gewünschte Adreßregister bezeichnet. Die Adreßregister sind 32 Bit breit. Im Gegensatz zum Datenregister kann der Zugriff auf Adreßregister nur mit Langworten (".L") oder Worten (".W") erfolgen. Wird ein Wort in ein Adreßregister übertragen, so wird daraus immer ein Langwort erzeugt, wobei auch ein Vorzeichenangleich durchgeführt wird.

Beispiel:

```
MOVEA.W #$8000,A0
```

Im Adreßregister A0 steht anschließend der Wert $FFFF8000. Datenregister verhalten sich nicht so, dort wird nur die angegebene Wortbreite im Register verändert.

Weiteres Beispiel:

```
MOVEA.L #$0000FFFF,A0    * Adreßregister belegen
ADDA.W #1,A0             * Wort addieren
```

In Register A0 steht anschließend der Wert $00010000. Trotz der Wortaddition erfolgte ein Übertrag in die zweite Worthälfte.

Beispiel für Datenregister:

```
        MOVE.L #$0000FFFF,D0    * Beispiel Datenregister
        ADD.W #1,D0             * Wort addieren
```

In D0 steht der Wert $00000000. Ein Überlauf ist nicht erfolgt.
Achtung: Das Adreßregister A7 ist für den Stackpointer reserviert.

(An)

Indirekte Adressierung:
Der Inhalt eines Adreßregisters (A0 bis A7) wird als Adresse interpretiert. Ein Zugriff auf Daten erfolgt dann mit Hilfe dieser Adresse.

Beispiel:

```
           LEA VARIABLE,A0      * Adresse VARIABLE
                                  nach A0
           MOVE.W (A0),D0       * Datenwert nach D0
           RTS
VARIABLE:  DC.W $1234
```

Die Adresse des Speicherplatzes mit dem Namen VARIABLE wird durch den LEA-Befehl in das Adreßregister A0 geladen. Der nachfolgende MOVE-Befehl transportiert ein Datenwort beginnend bei der in A0 angegebenen Adresse nach D0.

(An)+

Indirekte Adressierung mit Post-Increment (nachfolgende Erhöhung):
Wie bei (An) wird der Inhalt des Adreßregisters als Adresse interpretiert und auf den so angegebenen Operanden zugegriffen. Danach

wird aber der Inhalt des Adreßregisters erhöht. Um wieviel, hängt von der Wortlänge ab, die bei dem betreffenden Befehl angegeben wurde. Bei einem Byte-Zugriff ist das der Wert Eins, bei einem Wortzugriff wird um Zwei erhöht und bei einem Langwortzugriff um Vier.

Beispiel:

```
LOESCHEFELD: LEA FELD,A0             * Startadresse laden
             MOVE #ANZAHL-1,D3       * Anzahl laden
SCHLEIFE:    CLR.L (A0)+             * mit 0 belegen
             DBRA D3, SCHLEIFE       * wiederholen
             RTS

ANZAHL       EQU 100                 * Anzahl Langworte
FELD:        DS.L ANZAHL             * Platz reservieren
```

Zunächst wird das Adreßregister mit der Adresse von FELD belegt. Die Konstante ANZAHL gibt die Anzahl der Feldelemente an, wobei ein Feldelement ein Langwort breit sein soll.
Der Wert von ANZAHL wird als Schleifenzähler zum Beispiel in das Register D3 geladen.
In der Schleife wird zunächst mit CLR.L ein Langwort mit 0 belegt. Als Zieladresse wird der aktuelle Wert von A0 verwendet. Danach wird der Inhalt von A0 um 4 erhöht, da es sich um eine Langwortadressierung handelte. Der Befehl DBRA springt solange zur Marke SCHLEIFE, bis der Inhalt von D3 den Wert -1 angenommen hat, DBRA veringert den Inhalt von D3 nach jeder Ausführung um 1.

-(An)

Indirekte Adressierung mit Pre-Decrement (vorher Verringern):
Das angegebene Adreßregister wird zunächst erniedrigt. Die Anzahl ist durch die Wortbreite des Operanden bestimmt. Bei einem Byte-

Zugriff wird das Adreßregister um Eins verringert, bei einem Wortzugriff um Zwei und bei einem Langwortzugriff um Vier. Danach wird der neue Inhalt des Adreßregisters als Adresse interpretiert und auf den Operanden zugegriffen.

Beispiel:

```
MOVE.L D0,-(A7)        * D0 speichern
...
MOVE.L (A7)+,D0        * D0 zurück
...
```

Der Inhalt des Datenregisters D0 wird auf den Stack geladen. A7 ist beim 68xxx als Stackpointer reserviert. Da der Stack nach unten wächst, muß eine Decrementanweisung verwendet werden. Der Datenwert kann vom Stack durch eine umgekehrte Operation zurückgeholt werden.

d(An)

Indirekte Adressierung mit Displacement (Verschiebung):
Zum Inhalt des angegebenen Adreßregisters wird vor dem Bearbeiten des Operanden das angegebene Displacement addiert, und die so neu entstandene Adresse verwendet. Das Displacement kann nur eine 16-Bit Größe sein, die vor der Addition in eine 32-Bit-Zahl vorzeichengetreu umgewandelt wird. Zum Beispiel für das bearbeiten von Record-Strukturen eignet sich diese Adressierart ganz besonders.

Beispiel:

```
VAR1 EQU 0     * Wort Variable
VAR2 EQU 2     * Langwort
VAR3 EQU 6     * Wort
VLEN EQU 8
```

```
START:   LEA FELD,A0            * Anfang eines Felds
         MOVE.L VAR2(A0),D0     * z.B. VAR2 laden
         MOVE.W VAR3(A0),D1     * VAR3
         MOVE.W VAR1(A0),D2     * VAR1
         MOVE.W (A0),D3         * auch VAR1
         RTS
FELD:    DS.B VLEN              * Platz freihalten
```

Eine C-Definition sähe z.B. so aus:

```
struct feldart {
 short var1;
 long var2;
 short var3;
} feld;
```

Die Konstante VAR1 wird mit dem Wert 0 belegt, da dies die erste Variable in dem Rekord ist. VAR2 wird dann mit dem Wert von VAR1 plus der Größe von VAR1 in Bytes belegt, was 2 ergibt. VAR3 bekommt den Wert 6, da VAR2 + 4 = 6 ist und 4 Bytes durch VAR2 belegt werden.
VLEN schließlich gibt die gesamte Zahl der belegten Bytes an.

d(An,Xi)

Indirekte Adressierung mit Index und Displacement:
Der Inhalt des angegebenen Adreßregisters wird zum Inhalt des mit Xi angegebenen Registers addiert. Xi steht dabei für ein beliebiges Daten- oder Adreßregister und kann mit .W oder .L angegeben werden, also als 32-Bit-Größe oder 16-Bit-Größe interpretiert werden.
Bei einer 16-Bit-Größe wird der Wert vor der Addition auf eine 32-Bit-Größe vorzeichenrichtig erweitert. Nach der Addition wird das

Displacement aufaddiert, das eine 16-Bit-Größe sein kann, und ebenfalls vorzeichenrichtig zuvor auf eine 32-Bit-Zahl erweitert wird.
Diese Adressierart kann zum Beispiel für den Zugriff auf Feldvariablen verwendet werden.

Beispiel:

```
MOVE.W 5(A0,D1.L),D3
MOVE.L D4,$FFFF(A1,A2.W)
```

Beim ersten Befehl wird der Inhalt der Speicherzelle als Wort nach D3 transportiert, deren Adresse sich wie folgt errechnet: Inhalt von A0 als Langwort + Inhalt von D1 als Langwort + 5.

Beim zweiten Befehl wird das Langwort in D4 in eine Speicherzelle transportiert. Die Adresse ergibt sich aus der Summe des Inhalts von A1 + des vorzeichenerweiterten Inhalts von A2 (Wort) + minus 1, da $FFFF zuvor vorzeichenerweitert wird und $FFFFFFFF ergibt, also den Wert -1.

Abs.W

Absolute Adressangabe:
Eine 16-Bit-Größe wird als Adresse interpretiert. Dazu wird die Zahl zunächst vorzeichenerweitert auf 32-Bit angepaßt.

Beispiel:

```
MOVE.B $8000.W,D2
```

Der Inhalt der Speicherzelle $FFFF8000 wird in das Datenregister D0.B geladen.

Beispiel:

```
            MOVE.L VARIABLE.W,D3
            ...
VARIABLE:   DC.L $12345678
```

Der Inhalt des Speicherplatzes VARIABLE wird als Langwort in das Register D3 geladen. Dieser Befehl kann nur verwendet werden, wenn die absolute Adresse von VARIABLE im Bereich $0000 bis $7FFF oder $FFFF8000 bis $FFFFFFFF liegt.

Abs.L

Absolute Adressierung mit 32-Bit:
Die Adresse des Operanden wird direkt mit 32 Bit angegeben.

Beispiel:

```
            MOVE.L VARIABLE.L,D3
            ...
VARIABLE:   DC.L $12345678
```

Dieser Befehl kann im gesamten Adressraum des 68xxx arbeiten.

d(PC)

Programmzählerrelative Adressierung:
Das Displacement "d" gibt die relative Distanz zwischen dem aktuellen Programmzählerstand beim Befehl und der Adresse des Operanden an. Damit ist es möglich verschiebbare Programme zu erzeugen.

Beispiel:

```
        LEA TEXT(PC),A0
        JSR WRITE
        ...
TEXT:   DC.B "HALLO",0
```

Die meisten Assembler rechnen die relative Distanz automatisch aus, wenn man eine absolute Adresse wie hier z.B. TEXT angibt. Da die Distanz nur eine 16-Bit-Größe ist, muß der Operand im Bereich -32768 .. +32767 liegen, sonst ist er nicht erreichbar.
Es ist nicht möglich mit dieser Adressierart auch einen Zieloperand anzugeben, daher muß man zu einem Trick greifen, will man es dennoch tun:

```
          LEA VARIABLE(PC),A0  * abolute Adresse
                                 ermitteln
          MOVE (A0),D0         * dann damit arbeiten
          MOVE D0,(A0)
          ...
VARIABLE: DC.W 0
```

Hier wird durch den LEA-Befehl die absolute Speicheradresse ermittelt und in A0 abgelegt. Mit der indirekten Adressierung kann danach indirekt auf den Speicherplatz VARIABLE zugegriffen werden. Achtung: Schreibzugriffe in den Programmbereich können auch Probleme bei bestimmten Betriebssystemen verursachen, daher sollte man diesen Trick nur verwenden, wenn man darüber genau Bescheid weiß.

d(PC,Xi)

Programmzählerrelative Adressierung mit Indexregister:
Hier wird zusätzlich der Inhalt des angegebenen Adreßregisters addiert. Dabei kann das Indexregister ein Daten- oder Adreßregister

mit 16-Bit (.W) oder 32-Bit (.L) sein.

Beispiel:

```
        MOVE.L 10(PC,D0.W),D1
```

Der Inhalt des Programmzählers wird im auf 32-Bit erweiterten Inhalt des Datenregisters D0.W addiert und anschließend noch um 10 erhöht. Der so gebildete Wert wird als Adresse interpretiert und von der entsprechenden Speicherzelle wird ein Langwort geholt und in das Datenregister D1.L geladen.

Imm

Direkte Konstante (Abkürzung für Immediate):
Die Konstante wird durch Angabe des Zeichens "#" vom Assembler als solche erkannt. Beispiel:

```
        MOVE.W #10,ALPHA
```

Der Wert 10 wird als Wort nach ALPHA gespeichert. Achtung folgendes Beispiel zeigt die Bedeutung des Zeichens "#":

```
              MOVE.L VARIABLE,D0
              MOVE.L #VARIABLE,D0
              ...
VARIABLE:     ...
```

Im ersten Fall wird der Inhalt der Speicherzelle VARIABLE nach D0 geladen, im zweiten Beispiel wird die Adresse der Speicherzelle VARIABLE nach D0 geladen.

Adressierarten:

Assembler-code	Bedeutung
Dn	Datenregister direkt, D0..D7
An	Adreßregister direkt, A0..A7
(An)	Adreßregister indirekt, (A0)..(A7)
(An)+	Adreßregister indirekt mit nachfolgendem Increment des Adreßregisters
-(An)	Adreßregister indirekt mit vorausgehendem Decrement des Adreßregisters
d(An)	Adreßregister indirekt mit Offset (16 Bit)
d(An,Rx)	Adreßregister indirekt mit Indexregister und Offset (8 Bit)
wert.W	Absolute Adresse 16 Bit
wert.L	Absolute Adresse 32 Bit
d(PC)	Programmzähler relativ mit Offset (16 Bit)
d(PC,Rx)	Programmzähler relativ mit Index und Offset (8 Bit)
#konst	Direkte Konstante (8,16,32 Bit je nach Operandenangabe)

verwendete Abkürzungen:

cc	Bedingungscode (CC, CS, EQ, GE, GT, HI, LE, LS, LT, MI, NE, PL, VC, VS, T, F)
.x	Wortbreite: .B = Byte .W = Wort .L = Langwort
reglist	Registerliste, siehe MOVEM, z.B. D0-D3/A0-A6
wert,konst	Konstante

Dn,Dx,Dy	Datenregister D0..D7 erlaubt
An,Ax,Ay	Adreßregister A0..A7 erlaubt
Rn,Rx,Ry	Register, kann Daten- oder Adreßregister sein
marke	Zieladresse
<ea>	effektive Adresse, siehe Adressierarten

4 Befehlsreferenz

In diesem Kapitel finden Sie alle Befehle alphabetisch geordnet. Alle Befehle sind nach einem einfachen Schema aufgebaut:

Befehlscode
Direktkonstanten: Null, ein oder zwei Worte
Adreßmodeerweiterung für Quelloperand: Null, ein oder zwei Worte
Adreßmodeerweiterung für Zieloperand: Null, ein oder zwei Worte

Alle Befehle, die unterschiedliche Adressierarten zulassen, besitzen eine Adreßmodeangabe im Befehlscode, der in diesem Kapitel auch mit "ea" (effektive address) abgekürzt wird.

Adreßmode:

Daten-Register direkt	000 rrr
Adreß-Register direkt	001 rrr
Adreß-Register indirekt	010 rrr
Adreß-Register indirekt mit Postincrement	011 rrr
Adreß-Register indirekt mit Predecrement	100 rrr
Adreß-Register indirekt mit Displacement 16-Bit	101 rrr
	+16-Bit Wort mit Displacement
Adreß-Register indirekt mit Index und Displacement	110 rrr
	d/a ri ri ri w/l 0 0 0 d d d d d d d d

dabei gilt:
d/a für ri 0=Datenregister 1=Adreßregister
ri Index-Register 0..7
w/l 0=Vorzeichen erweitern (16-Bit Quelle), 1=Langwort
d Displacement 8-Bit wird vorzeichenerweitert.

Absolute Short 111 000
+ 16-Bit-Wort

Absolute Long 111 001
+ 32-Bit Langwort

Programm-Zähler indirekt mit
Displacement 111 101
+16-Bit Wort mit Displacement

Programm-Zähler indirekt mit
Index und Displacement 110 011
d/a ri ri ri w/l 0 0 0 d d d d d d d d

dabei gilt:
d/a für ri 0=Datenregister 1=Adreßregister
ri Index-Register 0..7 binär codiert
w/l 0=vorzeichenerweitern (16Bit Quelle), 1=Langwort
d Displacement 8-Bit wird vorzeichenerweitert.

Direkter Datenwert 111 100
+ 16-Bit Wort oder 32-Bit Wort
bei 8-Bit Datenkonstanten werden
die Bits 15 bis 8 mit 0 belegt.

Neben der Assemblersyntax ist auch das Befehlsformat für eine binäre Codierung der Befehle dargestellt. Ein Befehl (TST) mit dem Format

0 1 0 0 1 0 1 0 size size ea ea ea ea ea ea
size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
ea: Adreßmode

wird dabei wie folgt codiert:
Zunächst sind hier zwei Variable im Befehlscode vorhanden: size und ea. Size ist im Befehlsformat erläutert. Es soll z.B. der Code für TST.W $12345678.L erzeugt werden.
Für das Size-Feld wird dann der Code 01 verwendet, da es sich um eine Wort-Operation handelt. Als "ea" wird der Code 111 001 verwendet, da eine absolute Langwortadresse angegeben wurde. Der Maschinencode lautet dann:

0 1 0 0 1 0 1 0 0 1 1 1 1 0 0 1
0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0
0 1 0 1 0 1 1 0 0 1 1 1 1 0 0 0

oder sedezimal:

$4a79
$1234
$5678

ABCD

Bedeutung: Add Decimal with Extend
Addiere Dezimal mit X-Flag

Syntax: ABCD Dy,Dx
ABCD -(Ay),-(Ax)

Format: 1 1 0 0 rx rx rx 1 0 0 0 0 r/m ry ry ry

rx: Ziel-Register 0..7 (Datenregister wenn r/m =0, sonst Adreßregister)
r/m: =0 dann Datenregister
=1 dann Adreßregister mit Predecrement
ry: Quell-Register 0..7 (Datenregister wenn r/m =0, sonst Adreßregister)

Operand: Byte

Flags: N Z V C X
* ? * ? *

Z wird gelöscht, wenn das Ergebnis ungleich 0 ist, ansonsten bleibt es unverändert. Das Z-Flag muß daher vor Ausführung des Befehls auf 1 gesetzt werden, wenn man es später verwenden will.
Wenn mehrere Zahlen nacheinander addiert werden sollen, so liefert das Z-Flag am Schluß das Gesamtergebnis.

C wird bei einem dezimalen Übertrag gesetzt, sonst gelöscht.

X wie bei C-Flag.

N,Vundefiniert.

Mit dem Befehl kann man BCD-Zahlen addieren. BCD steht für Binär codierte Dezimalzahlen.
In einem Byte können dabei zwei Dezimalzahlen untergebracht werden.
Das X-Flag wird zusätzlich addiert und kann für eine Mehrfachaddition verwendet werden.

Beispiel:

Die Zahl 78 soll mit 34 addiert werden. Das Ergebnis soll in Register D1 stehen.

Das Programm lautet:

```
ADDIERE:     MOVE #4,CCR      * X-Flag löschen
             MOVE #$78,D0     * erster BCD-Wert
             MOVE #$34,D1     * zweiter BCD-Wert
             ABCD.B D0,D1     * Addition ausführen
             RTS              * Ende
```

Beispiel für mehrstellige Addition von BCD-Zahlen. Die Zahlen werden dazu im Speicher abgelegt.

```
ADDIEREMEHR: MOVE #4,CCR          * X-Flag löschen
             LEA ZAHL1,A0         * Adresse 1. Zahl
             LEA ZAHL2,A1         * Adresse 2. Zahl
             ABCD.B -(A0),-(A1)   * erstes Teilergebnis
             ABCD.B -(A0),-(A1)   * zweites Teilergebnis
```

```
           ABCD.B -(A0),-(A1)  * drittes Teilergebnis
           ABCD.B -(A0),-(A1)  * viertes Teilergebnis
           RTS

           DC.B $12            * Zahl 1 höchstwertigste Stelle
           DC.B $34
           DC.B $56
           DC.B $78
ZAHL1:     DC.B 0              * Ende der Zahl1 + 1

           DC.B $11            * Zahl 2 höchstwertige Stelle
           DC.B §22
           DC.B $33
           DC.B $44
ZAHL2:     DC.B 0              ; Ende der Zahl 2 + 1
```

Hier werden die Zahlen 12345678 mit 11223344 addiert. Das Ergebnis wird anstelle von Zahl 2 im Speicher abgelegt. Bei diesem Befehl muß darauf geachtet werden, das am Anfang die beiden Adreßregister um Eins erhöht anzugeben sind, denn bei der Befehlsausführung werden die Adreßregister zunächst decrementiert.

ADD

Bedeutung: Add Binary
Addiere Binär

Syntax: ADD.x <ea>,Dn
ADD.x Dn,<ea>

Format: 1 1 0 1 r r r opm opm opm ea ea ea ea ea ea

r: Register 0..7
opm: 000 Byte, Ziel Datenregister
001 Wort, Ziel Datenregister
010 Langwort, Ziel Datenregister
100 Byte, Ziel <ea>
101 Wort, Ziel <ea>
110 Langwort, Ziel <ea>
ea: Adreßmode

Operand: Byte, Wort, Langwort

Flags: N Z V C X
* * * * *

N wird gesetzt, wenn das Ergebnis negativ ist, sonst ge löscht.

Z wird gesetzt, wenn das Ergebnis Null ist sonst gelöscht.

V wird gesetzt, wenn ein Überlauf erzeugt wurde, sonst gelöscht.

C wird gesetzt, wenn ein Übertrag erzeugt wurde, sonst gelöscht.

X wie bei C-Flag.

Addition zweier Zahlen. Dabei kann eine Byte-Größe, eine Wort-Größe oder eine Langwort-Größe bearbeitet werden.

Beispiel:

```
START:    MOVE #4,D1         * Lade ersten Operanden
          MOVE #$100A,D2     * Lade zweiten Operanden
          ADD.W D1,D2        * Ergebnis nach D2
          RTS
```

Die Zahl 4 wird mit der Zahl $100A (dezimal 4106) addiert und das Ergebnis $100E (4110) in Register D2.W abgelegt.

Anstelle <ea> kann bei der Quelle stehen:

Dn
An (nur bei Wort - oder Langwort-Additionen)
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

und als Zielangabe:

(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

Wenn das Ziel ein Adreßregister sein soll, so muß man den Befehl ADDA ... verwenden. Man beachte, daß entweder das Ziel oder die Quelle ein Datenregister sein muß.

ADDA

Bedeutung: Add Address
Addiere Adresse

Syntax: ADDA.W <ea>,An
ADDA.L <ea>,An

Format: 1 1 0 1 r r r opm opm opm ea ea ea ea ea ea

r: Register 0..7
opm: 011 Wort Operation
111 Langwort Operation
ea: Adreßmode

Operand: Wort, Langwort

Flags: N Z V C X
\- - - - -

werden nicht verändert.

Der Quell-Operand wird zum Inhalt des angegebenen Adreßregisters addiert. Dabei wird immer das gesamte Adreßregister verändert, unabhängig ob ein Wort oder Langwort angegeben wurde.
Bei ADDA.W wird das Quellwort als vorzeichenbehaftete Zahl angesehen und vor der Addition in ein Langwort umgewandelt.

Beispiel:

```
START:    MOVE.W #$FFFF,D1     * auch -1
          LEA $20000,A0        * Adresse
          ADDA.W D1,A0         * Ergebnis nach A0
          RTS
```

Nach der Ausführung des Programms steht der Wert $1FFFF im Register A0.

Als Quelladresse <ea> sind möglich:

Dn
An (nur bei Wort - oder Langwort-Additionen)
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

ADDI

Bedeutung: Add Immediate
Addiere nachfolgenden Wert

Syntax: ADDI.x #konst,<ea>

Format:

```
0 0 0 0 0 1 1 0 s s ea ea ea ea ea ea
w w w w w w w w b b b b b b b b
l l l l l l l l l l l l l l l l
```

s: 00 Byte-Operation
01 Wort-Operation
10 Langwort-Operation
ea: Adreßmode
b: Bei einer Byte-Konstanten
w: Bei einem Wort
l: Bei einem Langwort

Operand: Byte, Wort, Langwort

Flags:

N	Z	V	C	X
*	*	*	*	*

N wird gesetzt, wenn das Ergebnis negativ ist, sonst gelöscht.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst gelöscht.

V wird gesetzt, wenn ein Überlauf erzeugt wurde, sonst gelöscht.

C wird gesetzt, wenn ein Übertrag erzeugt wurde, sonst gelöscht.

X, wie C-Flag.

Addition einer Konstanten zu einem Wert: Im Gegensatz zu den Befehlen ADD und ADDA kann hier auch eine Speicherzelle als Ziel mit der absoluten Adressierung angegeben werden.

Beispiel:

```
START:  ADDI.L #$1F23455A,ALPHA* nach Speicherzelle
        RTS

ALPHA:  DC.L $12220000
```

Hier wird der sedezimale Wert $1F23454A zum Wert $12220000 addiert.

Folgende Angaben sind als <ea> möglich:
- Dn
- (An)
- (An)+
- -(An)
- d(An)
- d(An,Xi)
- Abs.W
- Abs.L

ADDQ

Bedeutung: Add Quick
Addiere Schnell

Syntax: ADDQ.x #konst,<ea>

Format: 0 1 0 1 data data data 0 size size ea ea ea ea ea ea

data: 0..7 entspricht den Konstanten 8,1..7
size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
ea: Adreßmode

Operand: Byte, Wort, Langwort

Flags: X N Z V C
* * * * *

N wird gesetzt, wenn das Ergebnis negativ ist, sonst gelöscht.

Z wird gesetzt, wenn das Ergebnis Null iste sonst gelöscht.

V wird gesetzt, wenn ein Überlauf erzeugt wurde, sonst gelöscht.

C wird gesetzt, wenn ein Übertrag erzeugt wurde, sonst gelöscht.

X wie bei C-Flag.

Zahlen im Bereich 1 bis 8 können mit diesem Befehl direkt addiert werden. Der Befehl hat den Vorteil, daß er weniger Bytes im Speicher belegt und daher schneller ausgeführt werden kann. Wenn als Ziel ein Adreßregister angegeben wird, so ist eine Wort- oder Langwort-Addition zugelassen. Achtung: will man negative Zahlen addieren, verwendet man den Befehl SUBQ.

Beispiel:

```
START:    ADDQ #4,A5        * Adreßregister um 4 erhöhen
          RTS
```

Der Inhalt des Adreßregisters wird um 4 erhöht.

Folgende Zielangaben für <ea> sind möglich:

Dn
An (nur bei Wort - oder Langwort-Additionen)
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

ADDX

Bedeutung: Add Extend
Addiere mit X-Flag

Assembler: ADDX.x Dy,Dx
ADDX.x -(Ay),-(Ax)

Format: 1 1 0 1 rx rx rx 1 size size 0 0 r/m ry ry ry

rx: Ziel-Register 0..7 (Datenregister wenn r/m =0, sonst Adreßregister)
size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
r/m: =0 dann Datenregisterinhalt nach Datenregiste rinhalt
=1 dann Speicherinhalt nach Speicherinhalt indirekt über Adreßregister mit Predecre ment
ry: Quell-Register 0..7 (Datenregister wenn r/m =0, sonst Adreßregister)

Operand: Byte, Wort, Langwort

Flags:

X	N	Z	V	C
*	*	*	*	*

Z wird gelöscht, wenn das Ergebnis ungleich 0 ist sonst bleibt es unverändert. Das Z-Flag sollte daher bei nachfolgender Verwendung zuvor auf 0 gesetzt werden. Durch dieses Verhalten ist es möglich nach mehreren Additionen ein Gesamtergebnis für das Z-Flag zu erhalten.

C wird bei einem Übertrag gesetzt, sonst gelöscht.

X wie beim C-Flag.

N wird gesetzt, wenn das Ergebnis negativ ist, sonst gelöscht.

V wird gesetzt, wenn ein Überlauf stattfindet, sonst gelöscht.

Mit diesem Additions-Befehl können beliebig lange Daten aufsummiert werden, da das X-Flag mit addiert wird.

Beispiel:

```
START:  LEA ZAHL1,A0          * Adresse 1. Zahl
        LEA ZAHL2,A1          * Adresse 2. Zahl
        MOVE #4,CCR           * X-Flag löschen.
        ADDX.L -(A0),-(A1)    * erstes Langwort addieren
        ADDX.L -(A0),-(A1)    * zweites Langwort
        ADDX.L -(A0),-(A1)    * drittes Langwort
        ADDX.L -(A0),-(A1)    * viertes Langwort
        RTS

        DC.L $12345678        * höchstwertige Stelle Zahl 1
        DC.L $11223344
        DC.L $11112222
        DC.L $11111111
```

```
ZAHL1:   DS 0                 * zeigt auf nächste Speicherzelle
         DC.L $8888AAAA       * höchstwertige Stelle Zahl 2
         DC.L $44444444
         DC.L $33333333
         DC.L $22222222
ZAHL2:   DS 0                 * zeigt auf nächste Speicherzelle
```

Die Zahl $12345678112233441111222211111111 wird mit der Zahl $8888AAAA444444443333333322222222 addiert und das Ergebnis anstelle von Zahl2 im Speicher abgelegt.

AND

Bedeutung: AND Logical
Und-Verknüpfung

Syntax: AND.x <ea>,Dn
AND.x Dn,<ea>

Format: 1 1 0 0 r r r opm opm opm ea ea ea ea ea ea

r: Register 0..7
opm: 000 Byte, Ziel Datenregister
001 Wort, Ziel Datenregister
010 Langwort, Ziel Datenregister
100 Byte, Ziel <ea>
101 Wort, Ziel <ea>
110 Langwort, Ziel <ea>
ea: Adreßmode

Operand: Byte, Wort, Langwort

Flags: X N Z V C
\- * * 0 0

N wird gesetzt, wenn das höchstwertige Bit des Ergebnisses gesetzt ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis 0 ist, sonst rückgesetzt.

V,C werden immer rückgesetzt.

X unverändert.

Eine logische Und-Verknüpfung wird durchgeführt.

Beispiel:

```
START:   MOVE .L WERT1,D0    * Parameter 1
         AND.L D0,WERT2      * Ergebnis direkt nach Wert 2
         RTS

WERT1:   DC.L $0F0F0F05
WERT2:   DC.L $00FF00FE
```

Das Ergebnis steht in “WERT2” und lautet: $000F0004.
Als Quellangabe für <ea> sind folgende Adressierarten erlaubt:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

Als Zielangabe für <ea> sind möglich:

(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

ANDI

Bedeutung: AND Immediate
Und-Verknüpfung mit Konstanten

Syntax: ANDI.x #konst,<ea>

Format:

0	0	0	0	0	0	1	0	s	s	ea	ea	ea	ea	ea	ea
w	w	w	w	w	w	w	w	b	b	b	b	b	b	b	b
l	l	l	l	l	l	l	l	l	l	l	l	l	l	l	l

s: 00 Byte-Operation
01 Wort-Operation
10 Langwort-Operation
ea: Adreßmode
b: Bei einer Byte-Konstanten
w: Bei einem Wort
l: Bei einem Langwort

Operand: Byte, Wort, Langwort

Flags: N Z V C X
- * * 0 0

N wird gesetzt, wenn das höchstwertige Bit des Ergebnisses gesetzt ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis 0 ist, sonst rückgesetzt.

V,C werden immer rückgesetzt.

X unverändert.

Eine logische Und-Verknüpfung mit der angegebenen Konstanten wird durchgeführt.

Beispiel:

```
START:    ANDI.B #$3F,ALPHA * Ergebnis nach ALPHA
          RTS

ALPHA:    DC.B $75
```

In “ALPHA” steht anschließend der Wert $35.
Der Befehl wird auch verwendet um z.B. auf Werte von IO-Ports zu warten:

```
SCHLEIFE: ANDI.B #$10,$FFFFFF3A  * warten bis Port Bit 4
          BEQ.S SCHLEIFE         * auf 1 liegt
          RTS                    * dann beenden
```

Das Programm wartet solange, bis Bit 4 auf Adresse $FFFFFF3A den Wert 1 annimmt.

Als Zielangabe sind für <ea> gültig:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

ANDI to CCR

Bedeutung: And Immediate to Condition Code Register
Und- Verknüpfung mit dem
Bedingungscode-Register

Syntax: ANDI #konst,CCR

Format: 0 0 0 0 0 0 1 0 0 0 1 1 1 1 0 0
0 0 0 0 0 0 0 0 b b b b b b b b

b: Byte data

Operand: Byte

Flags: X N Z V C
* * * * *

X wird 0, wenn Bit 4 den Wert 0 hat, sonst unverändert.
N wird 0, wenn Bit 3 den Wert 0 hat, sonst unverändert.
Z wird 0, wenn Bit 2 den Wert 0 hat, sonst unverändert.
V wird 0, wenn Bit 1 den Wert 0 hat, sonst unverändert.
C wird 0, wenn Bit 0 den Wert 0 hat, sonst unverändert.

Mit diesem Befehl können einzelne Flags auf 0 gesetzt werden.

Beispiel:

```
LOESCHCARRY:        ANDI #%11110,CCR
                    RTS
```

Das C-Flag ist nach Ausführung des Befehls nicht gesetzt, also auf 0.

Zum Setzen von Flags wird der Befehl “ORI to CCR” verwendet.

ANDI to SR

Bedeutung: And Immediate to Status Register
Und-Verknüpfung mit Status-Register

Syntax: ANDI #konst,SR

Format: 0 0 0 0 0 0 1 0 0 1 1 1 1 1 0 0
w w w w w w w w w w w w w w w w
w: Datenwort

Operand: Wort

Flags: X N Z V C
* * * * *

Alle Bits des Statusregisters werden verändert. Sie werden auf 0 gesetzt, wenn das entsprechende Bit der Konstanten 0 ist.
Dieser Befehl ist privilegiert und arbeitet nur im System-Mode.

Die Belegung des Statusregisters:
- Bit 15 Trace-Mode
- Bit 13 Supervisor-State
- Bit 10 bis Bit 8 Interrupt-Level
- Bit 4 X-Flag
- Bit 3 N-Flag
- Bit 2 Z-Flag
- Bit 1 V-Flag
- Bit 0 C-Flag

Beispiel:

ANDI #%1111100011111111,SR

Dadurch wird der Interrupt-Level auf 0 gesetzt. Danach sind alle Interrupt-Ebenen freigeschaltet und alle Interrupts erlaubt.

ASL, ASR

Bedeutung: Arithmetic Shift
Arithmetisch schieben

Syntax:
ASL.x Dx,Dy
ASL.x #konst,Dn
ASL <ea>
ASR.x Dx,Dy
ASR.x #konst,Dn
ASR <ea>

Format: 1 1 1 0 c/r c/r c/r dr size size i/r 0 0 r r r
Register schieben

c/r Anzahl der Schiebeoperationen oder
Register 0..7
dr: =0 dann rechts schieben
=1 dann links schieben
size: 00 Byte-Operation
01 Wort-Operation
10 Langwort-Operation
i/r: =0, dann steht in c/r die Anzahl
=1, dann steht in c/r ein Register D0..D7
r: Register D0..D7 für das Ziel.

1 1 1 0 0 0 0 dr 1 1 ea ea ea ea ea ea
Speicher schieben

dr: =0 dann rechts schieben
=1 dann links schieben

ea: Adreßmode

Operand: Byte, Wort, Langwort

Flags: X N Z V C
* * * * *

Beim ASL-Befehl wird das höchstwertige Bit in das X- und in das C-Flag kopiert.
Beim ASR-Befehl wird Bit 0 in das X- und in das C-Flag kopiert.

Der ASL-Befehl schiebt einen Datenwert nach links. Das bedeutet Bit 0 kommt nach Bit 1, das alte Bit 1 nach Bit 2 usw. Bei einem arithmetischen Schiebebefehl wird der Wert 0 nachgeschoben, kommt also an die Stelle von Bit 0.
Der ASR-Befehl schiebt ein Datenwert nach rechts. Dabei wird also Bit 7 nach Bit 6 geschoben, Bit 6 nach Bit 5 usw. Wenn das höchstwertige Bit (31,15 oder 7 je nach Wortlänge) auf 0 war, so wird eine 0 nachgeschoben, war es auf 1, so wird entsprechend eine 1 nachgeschoben. Dadurch wird erreicht, daß eine vorzeichenbehaftete Zahl nach der Operation das Vorzeichen behält.

Beispiel:

```
START:  MOVE.W #$FFFC,D0  * Datenwert laden.
        ASR.W #1,D0       * um eine Stelle verschieben.
        RTS
```

$FFFC wird als Wort nach rechts geschoben. Nach der Ausführung

des Befehls steht in Register D0 der Wert $FFFE.
Die Konstante gibt die Anzahl der Schiebeoperationen an, so kann man mit ASR.W #2,D0 z.B. den Inhalt von D0.W um zwei Binärstellen nach rechts schieben.
Gibt man anstelle der Konstanten ein Register an, so wird der Inhalt des Registers als Anzahl von Schieboperationen interpretiert. Dabei ist der Bereich von 0..31 zuläßig. Wenn man als Ziel kein Register angibt, so kann man nur um eine Stelle schieben und es wird nur eine Bytegröße verschoben, Wort und Langwortangabe sind also nicht zulässig.

Weitere Beispiele:

```
START:  ASL WERT                * Inhalt von WERT nach
                                  links
        MOVE.L #$124F0A55,D0    * laden Wert
        ASL.L #2,D0             * 2 Stellen nach links
        MOVE #$8000,D1          * anderen Wert
        ASR.W #1,D1             * 1 Stelle nach rechts
        RTS

WERT:   DC.B $2A
```

Nach Ausführung des Programms steht in "WERT" der Betrag $54, in D0.L der Wert $493C2954 und in D1.W der Wert $C000.

Als Zielangabe können an Stelle von <ea> folgende Adressierarten verwendet werden:

(An)
(An)+
-(An)

d(An)
d(An,Xi)
Abs.W
Abs.L

Bcc

Bedeutung: Branch Conditionally
bedingter Sprung

Syntax: Bcc marke
Bcc.S marke

Format: 0 1 1 0 c c c c b b b b b b b b
w w w w w w w w w w w w w w w w
c: Condition-Code
b: Byte Offset
w: Wort Offset, wenn verwendet, dann
Byte Offset=0

Operanden: Byte, Wort

Flags: X N Z V C
\- - - - -

werden nicht verändert.

Ein relativer, bedingter Sprung: Die Zielangabe erfolgt im Maschinencode relativ zum aktuellen Programmzähler. Der Programmcode bleibt damit verschiebbar.

Der Sprung wird ausgeführt, wenn die angegebene Bedingung erfüllt ist. Folgende Bedingungen sind möglich:

CC 0100 carry clear mit C-Flag auf 0
CS 0101 carry set mit C-Flag auf 1
EQ 0111 equal mit Z-Flag auf 1
GE 1100 greater or equal, größer gleich mit N=1,V=1 oder N=0,V=0
GT 1110 greater than, größer als mit N=1,V=1,Z=0 oder N=0,V=0,Z=0
HI 0010 high, mehr als mit C=0,Z=0
LE 1111 less or equal, kleiner gleich mit Z=1 oder N=1,V=0, oder N=0, V=1
LS 0011 low or same, weniger als oder gleich mit C=1 oder Z=1
LT 1101 less than, kleiner als mit N=1,V=0 oder N=0,V=1
MI 1011 minus, negativ mit N=1
NE 0110 not equal, ungleich mit Z=0
PL 1010 plus, positiv mit N=0
VC 1000 overflow clear, Überlauf gelöscht mit V=0
VS 1001 overflow set, Überlauf gesetzt mit V=1

Der Sprungbefehl kann im Bereich +127,-127 (bei Angabe von .S) oder +32767,-32768 erfolgen. Im ersten Fall belegt der Befehlscode nur ein Wort, sonst zwei Worte.

Beispiel:

```
START:    MOVE ALPHA,D0  * zu prüfenden Wert laden
          BEQ ISTNULL    * wenn 0, dann springen
          ...            * sonst hier weiter
ISTNULL:  ...            * Einsprung bei 0
          RTS

ALPHA:    DC.W $1234
```

In diesem Beispiel wird der Sprung nicht ausgeführt. Achtung: Der MOVE-Befehl verändert die Flags. Bei anderen Prozessoren (z.B. 8086 oder Z80) ist das nicht der Fall.

Weiteres Beispiel:

```
...
CMP #100,D0
BLE KLEINERGLEICH
...
```

Die Programmausführung springt zur Marke KLEINERGLEICH, wenn der Inhalt von D0 kleiner oder gleich 100 ist. Der CMP-Befehl setzt die Flags für den Vergleich entsprechend. Hier noch ein kleiner Trick zur richtigen Verwendung der bedingten Sprünge bei Vergleichen zwischen zwei Operanden:

WENN operand1 <cc> operand2 DANN SPRINGE NACH SPRUNGZIEL

läßt sich übersetzen in:

```
CMP operand2,operand1
Bcc SPRUNGZIEL
```

wobei cc z.B. für LE, GE, EQ ... stehen kann.

Beispiel:

WENN D1 größer gleich D3 DANN SPRINGE NACH ZIELENDE

für "größer gleich" wird GE gesetzt und man erhält:

```
                CMP D3,D1
                BGE ZIELENDE
```

Siehe auch CMP für erlaubte Adressierarten.

Achtung: bei einem Sprungziel, das weiter als +32767 oder -32768 entfernt ist muß der absolute Sprung JMP verwendet werden. Wenn man eine bedingten Sprung mit einer großen Sprungweite ausführen will, so muß man zunächst zu einem JMP springen, der dann die große Entfernung überbrücken kann. Beispiel:

```
                BEQ NAHE
                ...
NAHE:           JMP WEITERSPRINGEN
```

Der JMP-Befehl arbeitet allerdings absolut, man kann damit keinen verschiebbaren Code erzeugen.

BCHG

Bedeutung: Test a Bit and Change
Prüfe ein Bit und wechsle

Syntax: BCHG Dn,<ea>
BCHG #konst,<ea>

Format: 0 0 0 0 r r r 1 0 1 ea ea ea ea ea ea Dn,<ea>

r: Register D0..D7
ea:

0 0 0 0 1 0 0 0 0 1 ea ea ea ea ea ea #konst,<ea>
0 0 0 0 0 0 0 0 bi bi bi bi bi bi bi bi

ea: Adreßmode
bi: Bitnummer 0..31 (modulo 32)

Operand: Byte, Langwort

Flags: X N Z V C
- - * - -

Z wird gesetzt, wenn das angegebene Bit auf 0 war, sonst rückgesetzt.

Der Befehl prüft ein Bit im Speicher oder Datenregister, der Wert des Bits wird anschließend komplementiert.
Wenn ein Datenregister angegeben wird, so können alle 32 Bit

geprüft werden, bei einer Speicherzelle ist nur die Byteadressierung zulässig, also Bit 0..7.
Die Bitnummer kann bei dem Befehl direkt angegeben werden oder indirekt in einem Register stehen.

Beispiel:

```
START:    BCHG #27,D3
          BEQ BITWARNULL
          MOVE #3,D4
          BCHG D4,ALPHA
          BNE BITWAREINS
          RTS
ALPHA:    DC.B $FF
```

Wenn Bit 27 im Register D4 Null war, so wird die Marke BITWARNULL angesprungen. Ferner wird der Inhalt von Bit 27 im Register D4 anschließend komplementiert, stand eine 1 dort, so befindet sich nachher eine 0 an der Bitposition 27 und umgekehrt.
War Bit 27 auf Eins, so wird die Speicherzelle "ALPHA" geprüft, wenn an Bitposition 3 (Inhalt von D4) in "ALPHA" eine 1 steht, so wird nach BITWAREINS gesprungen, wie es hier im Beispiel auch der Fall ist. Der Inhalt von ALPHA ist danach $F7, da Bit 3 nach dem BCHG-Befehl komplementiert wird.
Als Adreßangaben für <ea> sind zulässig:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

BCLR

Bedeutung: Test a Bit and Clear
Prüfe ein Bit und lösche

Syntax: BCLR Dn,<ea>
BCLR #konst,<ea>

Format: 0 0 0 0 r r r 1 1 0 ea ea ea ea ea ea Dn,<ea>

r: Register D0..D7
ea: Adreßmode

0 0 0 0 1 0 0 0 1 0 ea ea ea ea ea ea #konst,<ea>
bi bi bi bi bi bi bi bi bi bi bi bi bi bi bi bi

ea: Adreßmode
bi: Bitnummer 0..31 (modulo 32)

Operand: Byte, Langwort

Flags:

X	N	C	V	Z
-	-	*	-	-

Das Z-Flag wird gesetzt, wenn das angegebene Bit auf 0 war, sonst rückgesetzt.

Das angegebene Bit wird nach Ausführung des Befehls auf 0 gesetzt. Siehe auch BCHG.

Beispiel:

BCLR #0,$FFFFFFC0

Das Bit 0 auf Speicherzelle $FFFFFFC0 wird auf 0 gesetzt. Zuvor wird aber der alte Wert noch in das Z-Flag kopiert.
Bei Angabe eines Datenregisters als Ziel, sind alle 32 Bit adressierbar, sonst sind nur Byte-Ziele verwendbar.

Als <ea> sind erlaubt:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

BKPT *(nur 68010/68012)*

Bedeutung: Breakpoint
Unterbrechungsmarke

Syntax: BKPT #data

Format: 0 1 0 0 1 0 0 0 0 1 0 0 1 d d d

d: Wert 0..7

Operand: keiner

Flags: X N Z V C
\- - - - -

werden nicht verändert.

Dieser Befehl arbeitet nur auf dem 68010 und 68012. Dadurch wird ein breakpoint acknowledge bus cycle gestartet. Eine externe Hardware ist dann in der Lage, entweder den eigentlichen Befehl an den Prozessor zu senden oder die Ausführung zu unterbrechen. Durch die angegebene Konstante lassen sich 8 verschiedene Breakpoint z.B. von einem Debugger unterscheiden.

Der Befehl wird normalerweise von einem Debugger in das Programm gesetzt und der alte Befehl ausgetauscht, daher ist es nicht üblich den Befehl direkt in ein Programm zu schreiben.

BRA

Bedeutung:	Branch Always Springe immer
Syntax:	BRA marke BRA.S marke
Format:	0 1 1 0 0 0 0 0 b b b b b b b b w w w w w w w w w w w w w w w w b: Byte Offset w: Wort Offset, dann Byte Offset = 0
Operand	Byte, Wort
Flags:	X N Z V C - - * - - werden nicht beeinflußt.

Relativer unbedingter Sprung. Das Sprungziel wird im Maschinencode als relative Distanz zum Programmzähler angegeben. Das so erzeugte Programm ist verschiebbar.
Der Sprungbereich kann +127, -128 bei Angabe von ".S" betragen oder +32767, -32768 maximal. Der Befehl belegt nur ein Wort bei der kurzen Adressierart.

Beispiel:

```
START:      BRA WEITER
            ...
WEITER:     ...
```

BSET

Bedeutung: Test a Bit and Set
Prüfe ein Bit und setze

Syntax: BSET Dn,<ea>
BSET #konst,<ea>

Format: 0 0 0 0 r r r 1 1 1 ea ea ea ea ea ea Dn,<ea>

r: Register D0..D7
ea: Adreßmode

0 0 0 0 1 0 0 0 1 1 ea ea ea ea ea ea # konst, <ea>
bi bi bi bi bi bi bi bi bi bi bi bi bi bi bi bi

ea: Adreßmode
bit: Bitnummer 0..31 (modulo 32)

Operand: Byte, Langwort

Flags: X N Z V C
\- - * - -

Das Z-Flag wird gesetzt, wenn das angegebene Bit auf 0 war, sonst rückgesetzt.

Das angegebene Bit wird nach Ausführung des Befehls auf 1 gesetzt. Siehe auch BCHG, BCLR.

Beispiel:

BSET #31,D0

Das Bit 31 im Register D0 wird auf 1 gesetzt. Zuvor wird aber der alte Wert noch in das Z-Flag kopiert.
Bei Angabe eines Datenregisters als Ziel sind alle 32 Bit adressierbar, sonst sind nur Byte-Ziele verwendbar.

Als <ea> sind erlaubt:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

BSR

Bedeutung:	Branch to Subroutine Springe ins Unterprogramm
Syntax:	BSR marke BSR.S marke
Format:	0 1 1 0 0 0 0 1 b b b b b b b b w w w w w w w w w w w w w w w w
	b: Byte Offset w: Wort Offset, dann Byte Offset = 0
Operand:	Byte, Wort
Flags:	X N Z V C - - * - - werden nicht verändert.

Relativer, unbedingter Unterprogrammaufruf. Die Angabe der Zieladresse erfolgt im Maschinencode als relative Distanz zum aktuellen Programmzählerstand. Mit ".S" ergibt sich ein kürzerer Maschinencode, jedoch ist der Sprungbereich auf +127, -128 eingeschränkt. Ohne ".S" liegt der Bereich zwischen +32767 und - 32768. Will man weiter springen, so kann man den Befehl JSR verwenden, der allerdings absolut adressiert und keinen verschiebbaren Code erlaubt.

Bei Ausführung des Befehls wird der alte Inhalt des Programmzählers auf den Stack (A7) gelegt. Damit ist es möglich nach Ausführung des Unterprogramms mit RTS wieder an die alte Stelle zurückzukehren.

Beispiel:

```
HAUPTPRG:    BSR UNTERPRG1     * Aufruf Unterpro-
                                 gramm
             ...               * z.B. weitere Befehle
             BSR UNTERPRG1     * Nochmals Aufrufen
             ...               * usw.
             RTS               * ggf. auch Unterpro-
                                 gramm

UNTERPRG1:   ...               * diverse Befehle
             RTS               * Unterprogramm Ende
```

Nach Aufruf des HAUPTPROGRAMMS wird nach UNTERPROGRAMM gesprungen, und dort die Ausführung der Befehle fortgesetzt. Bei Ausführung des RTS-Befehls wird die Ausführung wieder im HAUPTPROGRAMM fortgesetzt. Ein erneuter Aufruf von UNTERPROGRAMM läßt wieder das Unterprogramm ausführen, nach dem Befehl RTS wird dann wieder das Hauptprogramm fortgesetzt. Das Hauptprogramm kann selbst auch wieder ein Unterprogramm sein und von einem anderen Programmteil aufgerufen worden sein.
Wichtig ist, daß Register A7, der Stackpointer mit einem gültigen Wert auf einen freien Speicherplatz im Ram zeigt, der für den Stack reserviert sein muß.

BTST

Bedeutung: Test a Bit
Prüfe ein Bit

Syntax: BTST Dn,<ea>
BTST Dn,#konst
BTST #konst,<ea>

Format: 0 0 0 0 r r r 1 0 0 ea ea ea ea ea ea Dn,<ea>
oder Dn,#konst

r: Register
ea: Adreßmode

0 0 0 0 1 0 0 0 0 0 ea ea ea ea ea ea #konst,<ea>
bi bi bi bi bi bi bi bi bi bi bi bi bi bi bi bi

ea: Adreßmode
bi: Bitnummer 0..31 (modulo 32)

Operand: Byte, Langwort

Flags: X N Z V C
- - * - -

Z wird gesetzt, wenn das geprüfte Bit auf 0 war, sonst rückgesetzt.

Das angegebene Bit wird nur geprüft und nicht verändert. Siehe auch BCHG,BCLR und BSET.

Im Gegensatz zu BCHG,BCLR und BSET besitzt dieser Befehl noch ein neues Befehlsformat:
BTST Dn,#konst. Als einziger Befehl steht hier die Konstante auf der rechten Seite. Der Befehl bedeutet dann: Prüfe die im Datenregister angegebene Nummer des Bits im Datenbyte, das auf der rechten Seite als Konstante angegeben ist.

Beispiel:

```
MOVE #5,D1
BTST D1,#%11011010
```

D1 enthält den Wert 5, BTST prüft dann Bit 5 der Konstanten %11011010. Hier ist das Bit auf 0. Sinnvoll ist dieser Befehl z.B. zur Realisation der Mengenabfragen, wie sie z.B. in PASCAL mit SET verwendet werden.

Achtung, manche Assembler verstehen diesen Befehl nicht, da er erst in neueren Unterlagen von Motorola beschrieben wird. Als <ea> sind zulässig:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)

CHK

Bedeutung: Check Register Against Bounds
Prüfe Registerinhalte gegen Grenzen

Syntax: CHK <ea>,Dn

Format: 0 1 0 0 r r r 1 1 0 ea ea ea ea ea ea

r: Register D0..D7
ea: Adreßmode

Operand: Wort

Flags:

X	N	Z	V	C
-	*	?	?	?

N wird gesetzt, wenn Dn < 0 und gelöscht, wenn Dn > (<ea>), sonst ist das Flag undefiniert.

Z,V,C undefiniert.

X bleibt unverändert.

Der Wortinhalt des angegebenen Datenregisters wird verglichen. Ist der Inhalt von Dn.W kleiner 0 oder größer als der Inhalt des angegebenen Wertes (Inhalt von <ea>), so wird eine CHK-Exception ausgelöst und der Programmfluß damit unterbrochen.
Achtung, es kann nur eine Wortgröße verwendet werden.

Beispiel:

CHK #10,D2

Wenn der Inhalt von D2 größer als 10 ist oder kleiner als 0, erfolgt eine CHK-Exception und damit eine Unterbrechung des Programms.

Als <ea> sind erlaubt:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

CLR

Bedeutung: Clear an Operand
Lösche ein Operand

Syntax: CLR.x <ea>

Format: 1 0 1 1 r r r opm opm opm ea ea ea ea ea ea

r: Register D0..D7
opm: 000 Byte-Operand
001 Wort-Operand
010 Langwort-Operand
ea: Adreßmode

Operand: Byte, Wort, Langwort

Flags:

X	N	Z	V	C
-	*	*	*	*

N wird gelöscht.

Z wird gesetzt.

V wird gelöscht.

C wird gelöscht.

X nicht verändert.

Alle Bits des angegebenen Ziels (Langwort, Wort oder Byte) werden auf 0 gesetzt. Damit lassen sich Speicherzellen oder Register mit 0 vorbelegen.

Beispiel:

```
                CLR.L ALPHA
                ...
ALPHA:          DS.L 1
```

Der Inhalt von "ALPHA" wird mit dem Wert $00000000 belegt.

Weiteres Beispiel:

```
START:     MOVE #1000,D1       * Anzahl der Speicher
                                 zellen
           LEA FELD,A0         * Zielfeld
SCHLEIFE:  CLR.L (A0)+         * löschen
           DBRA D1,SCHLEIFE
           RTS
FELD:      DS.L 1000           * 1000 Langworte
```

Der Inhalt von FELD wird mit 0 belegt, in diesem Fall sind das 1000 Langworte.

Achtung Hinweis:

Der CLR-Befehl führt beim 68000 und 68008 zunächst eine Lesezugriff auf die angegebene Speicherzelle durch. Verwendet man den CLR-Befehl zum Löschen von Port-Registern, kann das ggf. zu Störungen führen und man sollte den Befehl MOVE stattdessen verwenden.

Als Adressierarten für <ea> sind erlaubt:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

CMP

Bedeutung: Compare
Vergleiche

Syntax: CMP.x <ea>,Dn

Format: 1 0 1 1 r r r opm opm opm ea ea ea ea ea ea

r: Register D0..D7
opm: 000 Byte-Operand
001 Wort-Operand
010 Langwort-Operand
ea: Adreßmode

Operand: Byte, Wort, Langwort

Flags: X N Z V C
_ * * * *

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V wird gesetzt, wenn ein Überlauf erzeugt wird, sonst rückgesetzt.

C wird gesetzt, wenn ein Übertrag erzeugt wird, sonst rückgesetzt.

Der Datenwert der Quelle wird vom angegebenen Datenregister subtrahiert und die Flags werden entsprechend dem Result verändert. Das Ergebnis der Subtraktion wird nicht gespeichert.

Beispiel:

```
        CMP.L ALPHA,D3
        BGE GROESSERGLEICH
        ...
ALPHA:  DC.L  ...
```

Wenn der Inhalt von D3 größer oder gleich dem Inhalt von "ALPHA" ist, wird die Marke "GROESSERGLEICH" angesprungen. Siehe auch Bcc.

Als Adressierarten für <ea> sind erlaubt:

Dn
An (nur bei Wort - oder Langwort-Vergleichen)
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

CMPA

Bedeutung: Compare Address
Vergleiche Adresse

Syntax: CMPA.x <ea>,An

Format: 1 0 1 1 r r r opm opm opm ea ea ea ea ea ea

r: Register A0..A7
opm: 011 Wort-Operand
111 Langwort-Operand
ea: Adreßmode

Operand: Wort, Langwort

Flags:

X	N	Z	V	C
-	*	*	*	*

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V wird gesetzt, wenn ein Überlauf erzeugt wird, sonst rückgesetzt.

C wird gesetzt, wenn ein Übertrag erzeugt wird, sonst rückgesetzt.

Der Datenwert der Quelle wird vom angegebenen Adreßregister

subtrahiert und die Flags werden entsprechend dem Result verändert. Das Ergebnis der Subtraktion wird nicht gespeichert. Es dürfen nur Wort- oder Langwortvergleiche durchgeführt werden.

Beispiel:

```
START:      LEA START,A0          * Quell-Adresse laden
            LEA ZIEL,A1           * Ziel-Adresse laden
SCHLEIFE:   MOVE.B (A0)+,(A1)+    * Transport
            CMPA.L #ENDE,A0       * bis Ende erreicht
            BLT SCHLEIFE          * Sprung zurück
            RTS                   * Programmende

START:      DS.B 100              * Quelle
ENDE:       DS.B 0                * für Endevergleich
ZIEL:       DS.B 100              * Ziel
```

Solange A0 kleiner als 100 ist, wird die Marke "SCHLEIFE" angesprungen.
Als Adressierarten für <ea> sind erlaubt (nur Wort und Langwortangaben):

Dn
An
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

CMPI

Bedeutung: Compare Immediate
Vergleiche nachfolgend

Syntax: CMPI.x #konst,<ea>

Format:

0	0	0	0	1	1	0	0	s	s	ea	ea	ea	ea	ea	ea
w	w	w	w	w	w	w	w	b	b	b	b	b	b	b	b
l	l	l	l	l	l	l	l	l	l	l	l	l	l	l	l

s: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
ea: Adreßmode
b: Byte-Konstante
w: Wort-Konstante
l: Langwort-Konstante

Operand: Byte, Wort, Langwort

Flags:

X	N	Z	V	C
-	*	*	*	*

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V wird gesetzt bei Überlauf , sonst rückgesetzt.

C wird gesetzt bei Übertrag, sonst rückgesetzt.

Der angegebene Datenwert wird vom Inhalt des Ziels subtrahiert und die Flags werden entsprechend dem Ergebnis verändert. Das Ergebnis der Subtraktion wird nicht gespeichert.

Beispiel:

```
START:   MOVEA.L #$10000,A0    * Anfangsadresse
SUCHE:   CMPI.B #´A´,(A0)+     * Vergleich durchführen
         BNE SUCHE             * bis gefunden zurück
         RTS                   * Programmende
```

Beginnend bei der Adresse $10000 wird im Speicher nach dem ASCII-Zeichen ´A´gesucht. Wird es gefunden, so steht im Adreßregister A0 die nachfolgende Speicheradresse. Wird das Zeichen ´A´nicht gefunden, so endet dieses Programm niemals. In der Praxis müßt man daher z.B. einen Zähler mit einbauen, der die Suche auf eine bestimmte Anzahl begrenzt. Hinweis: das Programm würde in diesem Fall spätestens dann enden, wenn die Adressen des eigenen Programms durchsucht werden und im Maschinencode das Zeichen ´A´ gefunden wird, das im CMPI-Befehlscode stehen muß.

Folgende Adressierarten sind für <ea> zulässig:

- Dn
- (An)
- (An)+
- -(An)
- d(An)
- d(An,Xi)
- Abs.W
- Abs.L

CMPM

Bedeutung: Compare Memory
Vergleiche Speicher

Syntax: CMPM.x (Ay)+,(Ax)+

Format: 1 0 1 1 rx rx rx 1 size 0 0 1 ry ry ry

rx: Ziel-Register A0..A7
size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
ry: Quell-Register A0..A7

Operand: Byte, Wort, Langwort

Flags:

X	N	Z	V	C
-	*	*	*	*

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V wird gesetzt, wenn ein Überlauf erzeugt wird, sonst rückgesetzt.

C wird gesetzt, wenn ein Übertrag erzeugt wird, sonst rückgesetzt.

Sollen zwei Datenfelder miteinander verglichen werden, so kann dieser Befehl verwendet werden.
Dazu wird (Ax) von (Ay) subtrahiert. Die Flags werden entsprechend dem Ergebnis gesetzt. Anschließend werden die Adreßregister um Eins erhöht.

Beispiel:

```
              LEA TAB1,A0             * Anfangsadresse
                                        TAB1
              LEA TAB2,A1             * Anfangsadresse
                                        TAB2
              MOVE #MAXLEN-1,D1       * Länge der
                                        Tabelle-1
VERGLEICHE:   CMPM.B (A0)+,(A1)+      * Vergleich durch-
                                        führen
              BNE NICHTGLEICH         * nicht gleich, dann
                                        Sprung
              DBRA D1,VERGLEICHE      * Bis alles durch
                                        sucht.
              RTS                     * ok, alles gleich
                                        gewesen
NICHTGLEICH:  ...                     * nicht gleich,
                                        dann weiter

TAB1:         .... Datenwerte ....    * Tabelle 1
MAXLEN        EQU $-TAB1              * Länge der Tabelle
                                        festhalten

TAB2:         .... Datenwerte ....    * Tabelle 2 mit
                                        gleicher Anzahl
```

Die Datenwerte von TAB1 werden mit denen von TAB2 verglichen. Die Länge der Tabellen ist dabei in der Konstanten MAXLEN festgehalten. Wenn der Inhalt der beiden Tabellen nicht gleich ist, dann wird zur Marke NICHTGLEICH verzweigt. Wenn alle Elemente abgearbeitet sind, dann wird das Programm mit RTS beendet.

DBcc

Bedeutung: Test Condition, Decrement and Branch
Prüfe Bedingung und springe

Syntax: DBcc Dn,marke
DBRA Dn,marke

Format: 0 1 0 1 c c c c 1 1 0 0 1 r r r
w w w w w w w w w w w w w w w w

c: Condition Code
r: Register D0..D7
w: Wort Offset

Operand: Wort

Flags: X N Z V C
\- - - - -

Werden nicht verändert.

Der Befehl ist speziell für Schleifenkonstruktion gedacht. Falls ein Bedingungscode angegeben ist, so wird zunächst geprüft, ob die Bedingung erfüllt ist. Wenn ja, so wird der nachfolgende Befehl ausgeführt und die Schleife damit verlassen. Wenn nicht, dann wird das angegebene Datenregister (nur Wort) zunächst um Eins verringert. Ist der Wert ungleich -1, so wird die angegebene Marke angesprungen, sonst der nachfolgende Befehl ausgeführt.

Beispiel:

```
START:     LEA FELD1,A0            * erste Adresse
           LEA FELD2,A1            * zweite Adresse
           MOVE #1000-1,D3         * wegen -1 Prüfung
SCHLEIFE:  CMPM.B (A0)+,(A1)+      * Vergleich durchführen
           DBNE D3,SCHLEIFE        * bis Anzahl, oder ungleich
           ...                     * Auswertung
           RTS

FELD1:     DS.B 1000               * z.B. 1000 Elemente
FELD2:     DS.B 1000               * muß wie Feld1 sein
```

Die Schleife wird beendet, wenn entweder alle 1000 Werte verglichen wurden oder die Werte unterschiedlich waren. Anhand des Bedingungscode kann man anschließend feststellen, wodurch die Schleife beendet wurde. Mit BNE NICHTGEFUNDEN z.B. würde im Falle der Nichtübereinstimmung ein Sprung erfolgen.
Der Bcc-Befehl ist am ehesten mit einer REPEAT... UNTIL-Schleife vergleichbar, die jedoch über eine zusätzliche Zählmöglichkeit verfügt.

Condition Code:
CC 0100 carry clear mit C-Flag auf 0
CS 0101 carry set mit C-Flag auf 1
EQ 0111 equal mit Z-Flag auf 1
GE 1100 greater or equal, größer gleich mit N=1,V=1 oder N=0,V=0
GT 1110 greater than, größer als mit N=1,V=1,Z=0 oder N=0,V=0,Z=0
HI 0010 high, mehr als mit C=0,Z=0
LE 1111 less or equal, kleiner gleich mit Z=1 oder N=1,V=0, oder N=0, V=1

LS	0011 low or same, weniger als oder gleich mit C=1 oder Z=1
LT	1101 less than, kleiner als mit N=1,V=0 oder N=0,V=1
MI	1011 minus, negativ mit N=1
NE	0110 not equal, ungleich mit Z=0
PL	1010 plus, positiv mit N=0
VC	1000 overflow clear, Überlauf gelöscht mit V=0
VS	1001 overflow set, Überlauf gesetzt mit V=1

DIVS

Bedeutung: Signed Divide
Vorzeichenbehaftete Division

Syntax: DIVS <ea>,Dn

Format: 1 0 0 0 r r r 1 1 1 ea ea ea ea ea ea

r: Register D0..D7
ea: Adreßmode

Operand: Wort

Flags:

X	N	Z	V	C
-	*	*	*	0

N wird gesetzt, wenn der Quotient Null ist, sonst rückgesetzt. Bei einem Überlauf ist das Flag undefiniert.

Z wird gesetzt, wenn der Quotient Null ist, sonst rückgesetzt. Bei einem Überlauf ist das Flag undefiniert.

V wird gesetzt, wenn ein Überlauf entstand, sonst rückgesetzt.

C wird immer rückgesetzt.

Der Zieloperand (immer ein Datenregister) wird durch den Quellope-rand dividiert und das Ergebnis im Zieloperand abgespeichert. Dabei

ist der Dividend (Zieloperand) ein Langwort (32-Bit) und der Divisor ein Wort (16-Bit) breit. Das Ergebnis wird dann wie folgt zusammengesetzt:
In den Bits 0 bis 15 steht der Quotient und in den Bits 16 bis 31 steht der "Rest". Das Vorzeichen des Restes ist das Gleiche wie beim Quotient.
Bei einer Division durch Null wird eine DIV-Exception ausgelöst und damit das Programm unterbrochen. Bei einem Überlauf sind die Werte im Zielregister nicht gültig.

Beispiel:

```
DIVS #2,D1
```

Der Inhalt von D1.L wird durch 2 geteilt. Das Ergebnis der Division steht in D1.W und der Rest steht bei den Bitpositionen 16 bis 31. Wenn D1.L z.B. den Wert 5 vor der Division hatte, so steht nachher in D1.L der Wert $00010002. Dabei ist D1.W mit dem Ergebnis der Division, also dem Wert 2 belegt.
Hinweis, bei einer Division durch eine Zweierpotenz (also 2,4,8,...) ist es günstiger, da schneller, einen arithmetischen Schiebebefehl einzusetzen, hier also z.B. ASR #1,D1. Allerdings ergeben sich bei negativen Zahlen ggf. geringe Unterschiede: D1 = -1, bei DIVS #2,D1 erhält man 0 als Ergebnis, bei ASR #1,D1 den Wert -1.

Als Adressierarten für <ea> sind erlaubt:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W

Abs.L
d(PC)
d(PC,Xi)
#Imm

DIVU

Bedeutung: Unsigned Divide
Vorzeichenlose Division

Syntax: DIVU <ea>,Dn

Format: 1 0 0 0 r r r 0 1 1 ea ea ea ea ea ea

r: Register D0..D7
ea: Adreßmode

Operand: Wort

Flags:

X	N	Z	V	C
-	*	*	*	0

N wird gesetzt, wenn der Quotient Null ist, sonst rückgesetzt. Bei einem Überlauf ist das Flag undefiniert.

Z wird gesetzt, wenn der Quotient Null ist, sonst rückgesetzt. Bei einem Überlauf ist das Flag undefiniert.

V wird gesetzt, wenn ein Überlauf entstand, sonst rückgesetzt.

C wird immer rückgesetzt.

Der Zieloperand (immer ein Datenregister) wird durch den Quellope-rand dividiert und das Ergebnis im Zieloperand abgespeichert. Die Werte werden dabei als vorzeichenlose Zahl interpretiert. Dabei ist

der Dividend (Zieloperand) ein Langwort (32-Bit) und der Divisor ein Wort (16-Bit) breit. Das Ergebnis wird dann wie folgt zusammengesetzt:
In den Bits 0 bis 15 steht der Quotient und in den Bits 16 bis 31 steht der "Rest". Das Vorzeichen des Restes ist das Gleiche wie beim Quotient. Bei einer Division durch Null wird eine DIV-Exception ausgelöst und damit das Programm unterbrochen. Bei einem Überlauf sind die Werte im Zielregister nicht gültig.

Beispiel:

```
DIVU #10,D1
```

Der Inhalt von D1.L wird durch 10 geteilt. Das Ergebnis der Division steht in D1.W und der Rest steht bei den Bitpositionen 16 bis 31. Wenn D1.L z.B. den Wert 5 vor der Division hatte, so steht nachher in D1.L der Wert $00050000. Dabei ist D1.W mit dem Ergebnis der Division, also dem Wert 0 belegt.
Hinweis, bei einer Division durch eine Zweierpotenz (also 2,4,8,...) ist es günstiger, da schneller, einen logischen Schiebebefehl einzusetzen, hier also z.B. LSR #1,D1.

Als Adressierarten für <ea> sind erlaubt:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

EOR

Bedeutung: Exclusive OR Logical
Ausschließliche Oder-Verknüpfung

Syntax: EOR.x Dn,<ea>

Format: 1 0 1 1 r r r opm opm opm ea ea ea ea ea ea

r: Register D0..D7
opm: 100 Byte-Operand
101 Wort-Operand
110 Langwort-Operand
ea: Adreßmode

Operand: Byte,Wort,Langwort

Flags:

X	N	Z	V	C
-	*	*	0	0

N wird gesetzt, wenn das höchstwertige Bit des Ergebnisses gesetzt ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis 0 ist, sonst rückgesetzt.

V,C werden immer rückgesetzt.

Das angegebene Ziel wird mit dem Datenregisterinhalt Exclusiv-Oder-verknüpft und das Ergebnis wird im Ziel abgelegt.

```
START:     MOVE #%1100110000110011,D3   * erster
                                          Datenwert
           EOR.W D3,VARIABLE            * nach Ziel
           RTS

VARIABLE:     DC.W #%1010110010110101
```

Nach Aufruf enthält der Speicherplatz “VARIABLE” den Wert %0110000010000110.

Folgende Adreßangaben sind für <ea> möglich:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

EORI

Bedeutung: Exclusive OR Immediate
Verknüpfe ausschließlich mit nachfolgendem Wert

Syntax: EORI.x #konst,<ea>

Format:

0	0	0	0	1	0	1	0	s	s	ea	ea	ea	ea	ea	ea
w	w	w	w	w	w	w	w	b	b	b	b	b	b	b	b
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
ea: Adreßmode
b: Byte-Operand
w: Wort-Operand
l: Langwort-Operand

Operand: Byte, Wort, Langwort

Flags:

X	N	Z	V	C
-	*	*	0	0

N wird gesetzt, wenn das höchstwertige Bit des Ergebnisses gesetzt ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis 0 ist, sonst rückgesetzt.

V,C werden immer rückgesetzt.

Die angegebene Zahl wird mit dem Inhalt des Ziels Exclusiv-Oder-verknüpft und das Ergebnis wird im Ziel abgelegt.

```
START:        EORI.L #5,VARIABLE
              RTS

VARIABLE:     DC.W #$FF
```

Nach Aufruf enthält der Speicherplatz "VARIABLE" den Wert $FA.

Folgende Adreßangaben sind für <ea> möglich:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

EORI to CCR

Bedeutung: Exclusive OR Immediate to Condition Code Register
Verknüpfe ausschließlich, mit Bedingungsregister

Syntax: EORI #konst,CCR

Format: 0 0 0 0 1 0 1 0 0 0 1 1 1 1 0 0
0 0 0 0 0 0 0 0 b b b b b b b b

b: Byte Operand

Operand: Byte

Flags: X N Z V C
* * * * *

X wird komplementiert, wenn Bit 4 in der Konstanten 1 ist, sonst unverändert.
N wird komplementiert, wenn Bit 3 in der Konstanten 1 ist, sonst unverändert.

Z wird komplementiert, wenn Bit 2 in der Konstanten 1 ist, sonst unverändert.
V wird komplementiert, wenn Bit 1 in der Konstanten 1 ist, sonst unverändert.

C wird komplementiert, wenn Bit 0 in der Konstanten 1 ist, sonst unverändert.

Damit ist es möglich einzelne Flags zu komplementieren.

Beispiel:

```
START:      MOVE VARIABLE,D0
            EOR #4,CCR
            RTS

VARIABLE:   DC.W .... # $FF
```

Nach Aufruf enthält der Speicherplatz "VARIABLE" den Wert $FA.

Folgende Adreßangaben sind für <ea> möglich:

Dn
(An)
(An)+
d(An)
d(An,Xi)
Abs.w
Abs.l

Anschließend ist das Z-Flag gesetzt, wenn der Inhalt von VARIABLE ungleich Null war, sonst rückgesetzt.

EORI to SR

Bedeutung: Exclusive OR Immediate to the Status Register
Verknüpfe ausschließlich den nachfolgenden Wert mit dem Status-Register.

Syntax: EORI #konst,SR

Format:

0 0 0 0 1 0 1 0 0 1 1 1 1 1 0 0
w w w w w w w w w w w w w w w w

w: Wort Operand

Operand: Wort

Flags:

X N Z V C
* * * * *

Alle Bits des Statusregisters werden verändert. Sie werden komplementiert, wenn das entsprechende Bit der Konstanten 1 ist.
Dieser Befehl ist privilegiert und arbeitet nur im System-Mode.

Die Belegung des Statusregisters:

Bit 15	Trace-Mode
Bit 13	Supervisor-State
Bit 10 bis Bit 8	Interrupt-Level
Bit 4	X-Flag
Bit 3	N-Flag
Bit 2	Z-Flag
Bit 1	V-Flag
Bit 0	C-Flag

Beispiel: EORI #1,SR

Der Inhalt des Carry-Flags wird komplementiert. Anmerkung: Es ist schwer für diesen Befehl eine sinnvolle Anwendung zu finden. Er wird in der Praxis so gut wie nicht benötigt.

EXG

Bedeutung: Exchange Registers
Vertausche Register

Syntax: EXG Rx,Ry

Format: 1 1 0 0 rx rx rx 1 opm opm opm opm opm ry ry ry

rx: Daten- oder Adreßregister 0..7, bei einem Tausch zwischen Daten- und Adreßregister steht hier immer das Datenregister

opm: 01000 Datenregister Inhalte austauschen
01001 Adreßregister Inhalte austauschen
10001 Daten- mit Adreßregister Inhalten aus tauschen

ry: Daten - oder Adreßregister 0..7, bei einem Tausch zwischen Daten- und Adreßregister steht hier immer das Adreßregister

Operand: Langwort

Flags: X N Z V C
\- - - - -
werden nicht verändert.

Der Inhalt zweier Register wird miteinander vertauscht. Dabei werden immer alle 32-Bit verwendet, eine Längenangabe ist daher nicht erlaubt.

Es können Datenregister mit Datenregister, Datenregister mit Adreßregistern und Adreßregister mit Adreßregistern vertauscht werden.

Beispiel:

```
START:      MOVE.L #$12345678,D0
            LEA $abcdef01,A1
            EXG A1,D0
            RTS
```

Anschließend steht im Register D0.L der Wert $abcdef01 und im Register A1.L der Wert $12345678.

EXT

Bedeutung: Sign Extend
erweitere Vorzeichen

Syntax: EXT.x Dn

Format: 0 1 0 0 1 0 0 opm opm opm 0 0 0 r r r

opm: 010 Byte nach Wort vorzeichenerweitern
011 Wort nach Langwort vorzeichenerweitern

r: Datenregister D0..D7

Operand: Wort, Langwort

Flags:

X	N	Z	V	C
-	*	*	0	0

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V,C werden immer rückgesetzt.

Zulässig sind Wort und Langwortangaben. Eine vorzeichenbehaftete Zahl kann dabei von einer Wortgröße in einer nächsthöhere umgewandelt werden. Eine 8-Bit-Größe wird mit EXT.W vorzei-

chenrichtig in ein Wort umgewandelt und eine 16-Bit-Größe wird mit EXT.L in eine 32-Bit-Größe umgewandelt. Ist die Zahl zuvor positiv, so werden die neu hinzukommenden Bits mit Null aufgefüllt, wenn sie negativ ist, mit Einsen.

Beispiel:

```
MOVE.B #$80,D0        * Wert -128
EXT.W D0
EXT.L D0
RTS
```

Danach steht in Register D0.L der Wert $FFFFFF80.
Wäre D0.B zuvor mit $7F belegt (127), dann stünde anschließend die Zahl $0000007F in D0.L.

ILLEGAL

Bedeutung: Illegal Instruction
Unerlaubter Befehl

Syntax: ILLEGAL

Format: 0 1 0 0 1 0 1 0 1 1 1 1 1 1 0 0

Operand: keiner

Flags: X N Z V C
- - - - -
werden nicht verändert.

Es wird eine ILLEGAL-Exception ausgelöst und die normale Programmausführung unterbrochen. Mit diesem Befehl ist es z.B. möglich neue Befehle nachzubilden. Es führen auch andere Maschinencodes zu dieser ILLEGAL-Exception, die jedoch für die Emulation (Nachbildung) neuer Befehle nicht verwendet werden sollen.

Beispiel:

```
ILLEGAL   * eigener Spezialbefehl
DC.W ...  * z.B. Parameter für den Befehl
```

Im Exception-Unterprogramm muß man dann ggf. die Parameter selbst abholen und verarbeiten. Mit einem RTE-Befehl kehrt man wieder ins Hauptprogramm zurück, wobei man bei diesem Beispiel zuvor noch den Programmzählerstand korrigieren müßte.

JMP

Bedeutung: Jump
Springe

Syntax: JMP <ea>

Format: 0 1 0 0 1 1 1 0 1 1 ea ea ea ea ea ea
ea: Adreßmode

Operand: ohne Dimension

Flags: X Z N V C
- - - - -
werden nicht verändert.

Es wird ein unbedingter Sprung ausgeführt. Die angegebene Adresse ist dabei absolut zu sehen.

Beispiel:
```
LEA ZIEL,A0
JMP (A0)   * indirekter Sprung nach Ziel
```

Oder besser:
```
JMP ZIEL   * mit direkter Adreßangabe.
```

Als Adressierarten für <ea> sind zugelassen:

(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)

JSR

Bedeutung: Jump to Subroutine
Springe ins Unterprogramm

Syntax: JSR <ea>

Format: 0 1 0 0 1 1 1 0 1 0 ea ea ea ea ea ea
ea: Adreßmode

Operand: ohne Dimension

Flags: X N Z V C
\- - - - -
werden nicht verändert.

Ein Unterprogramm-Aufruf wird ausgeführt. Die Sprungzielangabe wird als absolute Adresse interpretiert. Bei Aufruf wird die Rückkehradresse auf den Stack (definiert durch A7) geschrieben. Siehe auch BSR-Befehl.

Das Unterprogramm wird normalerweise durch einen RTS-Befehl abgeschlossen.

Beispiel:

```
HAUPTPRG:   JSR UNTERPRG        * Aufruf
            ...                 * weitere Befehl

UNTERPRG:   ...                 * Befehle
            RTS                 * Rückkehr
```

Folgende Adressierarten können für <ea> verwendet werden:

(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)

LEA

Bedeutung: Load Effective Address
Lade verwendete Adresse

Syntax: LEA <ea>,An

Format: 0 1 0 0 r r r 1 1 1 ea ea ea ea ea ea

r: Adreßregister A0..A7
ea: Adreßmode

Operand: Langwort

Flags: X N Z V C
- - - - -
werden nicht verändert.

Die angegebene Adresse wird in ein Adreßregister geladen. Dabei wird je nach Adressierart die gültige Adresse zunächst berechnet.

Beispiel:

```
        LEA BUFFER(PC),A0
        ...
BUFFER: ...
```

Die Adresse von "BUFFER" wird nach A0 geladen. Dabei wird die Adresse relativ zum Programmzählerstand verwendet, so daß der erzeugte Programmcode verschiebbar bleibt.
Mit MOVEA.L #BUFFER,A0, stünde die absolute Adresse von

"BUFFER" im Maschinencode, und wenn der Code samt "BUFFER" verschoben wird, müßte man den Maschinencode mit der neuen Adresse belegen.
Folgende Adressierarten können anstelle von <ea> verwendet werden:

(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)

LINK

Bedeutung: Link and Allocate
Binde und reserviere

Syntax: LINK An,#konst

Format: 0 1 0 0 1 1 1 0 0 1 0 1 0 r r r
d d d d d d d d d d d d d d d d

r: Adreßregister A0..A7
d: Displacement für Stack

Operand: ohne Dimension

Flags: X N Z V C
\- - - - -
werden nicht verändert.

Soll ein Unterprogramm mit lokalen Variablen arbeitet, so ist es möglich Platz dafür auf dem Stack zu reservieren. Der Befehl kopiert dazu zunächst den alten Inhalt des angegebenen Adreßregisters auf den Stack, dann wird das Adreßregister mit dem Inhalt von A7 belegt, also dem Stackpointerwert. Danach wird die angegebene Konstante (16-Bit-Größe) auf den Inhalt von A7 (Stackpointer) addiert. Normalerweise muß eine negative Zahl als Konstante angegeben werden, da der Stack stets abwärts wächst. Durch indirekte Adressierung über das angegebene Adreßregister ist es möglich auf den Stack zuzugreifen und dort seine Variablen zwischenzuspeichern.

Beispiel:

Zwei Langworte sollen auf dem Stack als lokale Daten gespeichert werden. Das Register A6 soll als Pointer verwendet werden.

```
UPRG:   LINK A6,#-8             * Platz für zwei Langworte
        ...
        MOVE.L -4(A6),D0        * z.B laden des zweiten Wertes
        MOVE.L D0,-8(A6)        * abspeichern im ersten Wert
        ...
        UNLK A6                 * Platz freigeben
        RTS                     * Unterprogramm Ende
```

Gerade bei rekursiven Programmen, also z.B. Unterprogramme die sich selbst aufrufen können, ist diese Programmiertechnik ganz praktisch, denn bei jedem erneuten Aufruf wird auch neuer Speicherplatz reserviert. Auch ist diese Methode speicherplatzsparend, denn der Speicher wird nur belegt, wenn er auch gebraucht wird.

LSL, LSR

Bedeutung: Logical Shift
Logisches Schieben

Syntax: LSL.x Dx,Dy
LSL.x #konst
LSL.x <ea>
LSR.x Dx,Dy
LSR.x #konst
LSR.x <ea>

Format: 1 1 1 0 c/r c/r c/r dr size size i/r 0 1 r r r
Register schieben

c/r: wenn l/r=0, dann steht die Anzahl der Schiebeoperationen direkt da, dabei gilt 0,1..7 enstricht 8,1..7 Schiebevorgängen.
wenn l/r=1, dann steht dort das Datenregister D0..D7

dr: 0=Links schieben
1=Rechts schieben

size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand

i/r: =0, dann Konstante angegeben
=1, dann indirekt schieben, Register angegeben

r: Datenregister, dessen Inhalt geschoben wird

1 1 1 0 0 0 1 dr 1 1 ea ea ea ea ea ea **Speicher** schieben

dr: 0=Rechts schieben
1=Links schieben
ea: Adreßmode

Operand: Byte, Wort, Langwort

Flags:

```
X N Z V C
* * * 0 *
```

Beim LSL-Befehl wird das höchstwertige Bit in das X- und in das C-Flag kopiert.
Beim LSR-Befehl wird Bit 0 in das X- und in das C-Flag kopiert.

Der LSL-Befehl schiebt einen Datenwert nach links, Bit 0 kommt nach Bit 1, das alte Bit 1 nach Bit 2 usw. Bei einem logischen Schiebebefehl wird immer der Wert 0 nachgeschoben.
Der LSR-Befehl schiebt einen Datenwert nach rechts. Dabei wird also Bit 7 nach Bit 6 geschoben, Bit 6 nach Bit 5 usw. Es wird immer eine 0 nachgeschoben.

Beispiel:

```
START:  MOVE.W #$FFFC,D0   * Datenwert laden.
        LSR.W #1,D0        * um eine Stelle verschieben.
        RTS
```

$FFFC wird als Wort nach rechts geschoben. Nach der Ausführung des Befehls steht in Register D0 der Wert $7FFE.
Die Konstante gibt die Anzahl der Schiebeoperationen an, so kann man mit LSR.W #2,D0 z.B. den Inhalt von D0.W um zwei Binärstellen nach rechts schieben.

Gibt man anstelle der Konstanten ein Register an, so wird der Inhalt des Registers als Anzahl von Schieboperationen interpretiert. Dabei ist der Bereich von 0..31 zuläßig. Wenn man als Ziel kein Register angibt, so kann man nur um eine Stelle schieben und es wird nur ein Byte verschoben, Wort und Langwortangabe sind also nicht zulässig.

Weitere Beispiele:

```
START:   LSL WERT             * Inhalt von WERT
                              nach links
         MOVE.L #$124F0A55,D0 * laden Wert
         LSL.L #2,D0          * 2 Stellen nach links
         MOVE #$8000,D1       * anderen Wert
         LSR.W #1,D1          * 1 Stelle nach rechts
         RTS

WERT:    DC.B $2A
```

Nach Ausführung des Programms steht in "WERT" der Betrag $54, in D0.L der Wert $493C2954 und in D1.W der Wert $4000.

Als Zielangabe können an Stelle von <ea> folgende Adressierarten verwendet werden:

(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

MOVE

Bedeutung: Move Data from Source to Destination
Transportiere Daten von Quelle nach Ziel

Syntax: MOVE.x <ea>,<ea>

Format: 0 0 s s d d d d d d sea sea sea sea sea sea

s: 01 Byte-Operand
11 Wort-Operand
10 Langwort-Operand
d: Adreßmode für Ziel
sea: Adreßmode für Quelle

Operand: Byte, Wort, Langwort

Flags: X N Z V C
- * * 0 0

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.
Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.
V,C werden rückgesetzt.
X wird nicht verändert.

Der erste Operand (links) wird zum angegebenen Ziel (rechts) kopiert. Byte-, Wort- und Langworttransporte sind möglich.

Beispiel:

```
START:     LEA QUELLE,A0          * Pointer auf Quelle
           LEA ZIEL,A1            * Pointer auf Ziel
           MOVE #LAENGE-1,D1      * Zähler
SCHLEIFE:  MOVE.L (A0)+,(A1)+     * transport Langworte
           DBRA D1,SCHLEIFE       * LAENGE
                                  * Langwort
           RTS                    * bis fertig
```

Hier wird der Bereich beginnend bei "QUELLE" nach "ZIEL" kopiert, wobei die Felder "LAENGE" Langworte groß sind.

für den MOVE-Befehl sind folgende Adressierarten für <ea> als Quelle möglich:

Dn
An (nur bei Wort - oder Langwort-Zugriffen)
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

und als Zielangabe sind anstelle von <ea> möglich:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

Für den Transport in ein Adreßregister verwendet man entweder MOVEA oder LEA.

MOVEA

Bedeutung:	Move Address Transportiere Adresse
Syntax:	MOVEA.x <ea>,An
Format:	0 0 size size r r r 0 0 0 ea ea ea ea ea ea
	size: 11 Wort-Operand 10 Langwort-Operand
	r: Adreßregister A0..A7 ea: Adreßmode
Operand:	Wort, Langwort
Flags:	X N Z V C - - - - -

werden nicht verändert.

Der Inhalt der angegebenen Quelle (erster Operand) wird in ein Adreßregister kopiert. Anstelle von ".x" darf nur ".W" und ".L" verwendet werden, also Wort oder Langwort.
Im Gegensatz zu MOVE werden bei diesem Befehl die Flags nicht verändert.
Bei Transport eines Wortes, wird Vorzeichen automatisch erweitert und in ein Langwort umgewandelt.

Beispiel:

```
START:    MOVEA.W #$8000,A0
          MOVEA.L #$8000,A1
          RTS
```

Anschließend steht in A0 der Wert $FFFF8000 und in A1 der Wert $00008000.

Bei der Angabe von <ea> sind folgende Adressierarten erlaubt:

Dn
An (nur bei Wort - oder Langwort-Zugriffen)
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

-

MOVE from CCR (nur bei 68010 und 68012)

Bedeutung: Move from Condition Code Register
Transport aus Condition Code Register

Syntax: MOVE CCR,<ea>

Format: 0 1 0 0 0 0 1 0 1 1 ea ea ea ea ea ea
ea: Adreßmode

Operand: Wort

Flags: X N Z V C
\- - - - -
werden nicht verändert.

Der Inhalt des Condition-Code Registers wird in das angegebene Ziel geladen. Dieser Befehl ist beim 68010 und 68012 nötig geworden, da MOVE from SR dort privilegiert ist und nur im System-Mode ausgeführt wird.
Beispiel: MOVE CCR,D0

Als Adressierarten können verwendet werden:

- Dn
- (An)
- (An)+
- -(An)
- d(An)
- d(An,Xn)
- abs.W
- abs.L

MOVE to CCR

Bedeutung: Move to Condition Codes
Transport ins Bedingungscode Register

Syntax: MOVE <ea>,CCR

Format: 0 1 0 0 0 1 0 0 1 1 ea ea ea ea ea ea
bei MOVE to CCR

ea: Adreßmode

Operand: Wort

Flags: X N Z V C
* * * * *

X wird 0, wenn Bit 4 den Wert 0 hat, sonst unverändert.
N wird 0, wenn Bit 3 den Wert 0 hat, sonst unverändert.
Z wird 0, wenn Bit 2 den Wert 0 hat, sonst unverändert.
V wird 0, wenn Bit 1 den Wert 0 hat, sonst unverändert.
C wird 0, wenn Bit 0 den Wert 0 hat, sonst unverändert.

Beispiel für den MOVE to CCR-Befehl. Damit lassen sich einzelne Flags setzen, z.B.:

```
MOVE #1,CCR
```

Das C-Flag wird gesetzt, alle anderen Flags werden rückgesetzt.

Als Adressierarten können verwendet werden:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

MOVE from SR

(Achtung, beim 68010,68012 nur im System-Mode)

Bedeutung: Move from Status Register
Transport aus dem Status-Register

Syntax: MOVE SR,<ea>

Format: 0 1 0 0 0 0 0 0 1 1 ea ea ea ea ea ea
ea: Adreßmode

Operand: Wort

Flags: X N Z V C
\- - - - -
beim MOVE from SR-Befehl:
Flags werden nicht verändert.

Achtung: MOVE from SR ist beim 68010 und 68020 ein privilegierter Befehl und führt daher dort im User-Mode zu einer Exception. Dies ist der Grund daß viele Softwarepakete, die auf dem 68000 entwickelt wurden, auf dem 68010 oder 68020 nicht mehr arbeiten. Man sollte den Befehl möglichst nicht verwenden oder garantieren, daß er im System-Mode abläuft, z.B. durch den Umweg über einen Trap-Aufruf.

Beispiel:

```
MOVE SR,D0
```

D0.L wird mit dem Inhalt des Status-Registers geladen.

Als Adressierarten können verwendet werden:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

MOVE to SR

Bedeutung: Move to the Status Register
Transport in das Status-Register

Syntax: MOVE <ea>,SR

Format: 0 1 0 0 0 1 1 0 1 1 ea ea ea ea ea ea

ea: Adreßmode

Operand: Wort

Flags: X N Z V C
* * * * *

X wird 0, wenn Bit 4 den Wert 0 hat, sonst unverändert.
N wird 0, wenn Bit 3 den Wert 0 hat, sonst unverändert.
Z wird 0, wenn Bit 2 den Wert 0 hat, sonst unverändert.
V wird 0, wenn Bit 1 den Wert 0 hat, sonst unverändert.
C wird 0, wenn Bit 0 den Wert 0 hat, sonst unverändert.

Achtung, dieser Befehl ist privilegiert.

Beispiel:

```
MOVE #0,SR      * rücksetzen
```

Als Adressierarten können verwendet werden:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

MOVE USP

Bedeutung: Move User Stack Pointer
Transportiere User-Stack Pointer (A7)

Syntax: MOVE USP,An
MOVE An,USP

Format: 0 1 0 0 1 1 1 0 0 1 1 0 d r r r r
dr: 0=Adreßregister nach USP
1=USP nach Adreßregister
r: Adreßregister A0..A7

Operand: Langwort

Flags: X N Z V C
\- - - - -
Flags werden nicht verändert.

Der Befehl ist privilegiert und kann nur im System-Mode verwendet werden. Er transportiert den Inhalt des User-Stack-Pointers in ein Adreßregister und umgekehrt.

Beispiel:

```
        LEA FREI,A0      * Stack Ende
        MOVE A0,USP      * In User-Stackpointer laden
        ...
        DS.L 100
FREI:   DS 0
```

Hier wird der User-Stackpointer mit einem Wert geladen.

MOVEC *(nur 68010, 68012)*

Bedeutung: Move Control Register
Transportiere Kontroll-Register

Syntax: MOVEC Rc,Rn
MOVEC Rn,Rc
wobei Rc : SFC,DFC,USP oder VBR sein kann
und Rn: Adreß- oder Datenregister

Format:

0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	dr
a/d	r	r	r	c	c	c	c	c	c	c	c	c	c	c	c

dr: 0= Kontrollregister nach Register
1= Register nach Kontrollregister
a/d: 0= Datenregister
1= Adreßregister
r: Register D0..D7, oder A0..A7
c: 000000000000 SFC Register
000000000001 DFC Register
100000000000 USP Register
100000000001 VBR Register

Operand: Langwort

Flags: X N Z V C
\- - - - -
werden nicht verändert.

Der Befehl ist privilegiert und arbeitet daher nur im System-Mode. Im User-Mode erfolgt eine Exception. Der Inhalt des angegebenen Kontrollregisters wird in ein Arbeitsregister geladen oder umgekehrt.

Beispiel:

```
MOVE.L #1,D0
MOVEC D0,SFC
```

MOVEM

Bedeutung: Move Multiple Registers
Transportiere mehrere Register

Syntax: MOVEM.L reglist,<ea>
MOVEM.L <ea>,reglist

Format: 0 1 0 0 1 dr 0 0 1 Sz ea ea ea ea ea ea
m m m m m m m m m m m m m m m m
dr: 0=Register nach Speicher
1=Speicher nach Speicher
Sz: 0=Wort-Operand
1=Langwort-Operand
ea: Adreßmode
m: Register Liste
Bit 15..0: A7..A0,D7..D0 bei Postincrement
Bit 15..0: D0..D7,A0..A7 bei Predecrement

Operand: Wort,Langwort

Flags: X N Z V C
\- - - - -

werden nicht verändert.

Wort und Langwortadressierung sind möglich. Der Inhalt der angegebenen Registerliste wird bei der angegebenen Zieladresse abgelegt.
Die Registerliste besitzt ein neues Assemblerformat:

Mehrere Register lassen sich durch "/" getrennt angeben, oder durch ein " - " - Zeichen ist es möglich einen Registerbereich anzugeben.
Beispiel:

D0/D3/A2-A6

Dadurch werden die Register D0, D3, A2, A3, A3, A5, A5 und A6 angegeben.
Wird als Ziel eine Speicheradresse angegeben, so werden die angegebenen Register entweder als Wort oder als Langwort in aufeinanderfolgendenen Adressen dort abgelegt.

Beispiel:

```
START:          MOVEM.L D0-D6/A3,MERKEALLES
                RTS
MERKEALLES:     DS.L 8              * Platz für 8 Register
```

Wird als Ziel der Predecrementmode angegeben, so werden die Register auf vorher liegenden Adressen untergebracht (Stackadressierung).

Beispiel:

```
MOVEM.L D0-D4/A1-A4,-(A7)
```

Hier wird der Standard-Stackpointer verwendet. In den Stack werden dann die angegebenen Register gespeichert.
Die Werte können natürlich auch wieder zurückgeholt werden:

```
MOVEM.L (A7)+,D0-D4/A1-A4
```

Wichtig ist, daß beim zurückholen genau die gleichen Register angegeben werden, damit der Stackpointer dann wieder auf dem gleichen Stand ist wie vor dem Abspeichern.

Beispiel:

```
UPRG: MOVEM.L D0-D7,A0-A6,-(A7)   * alle Register retten
      ...                         * weitere Befehle
      MOVEM.L (A7)+,D0-D7/A0-A6   * alle Register zurück
      RTS
```

Ein typisches Unterprogramm. Alle Register werden zu Beginn gerettet und vor Beendigung zurückgespeichert. Das Register A7 kann dabei natürlich nicht in der Liste sein, denn es dient als Stackpointer.
Wird eine feste Adresse als Quelle angegeben, so werden die angegebenen Register genau in umgekehrter Reihenfolge wie beim Speichern in die angegebenen Register zurückgeholt. Beim Autoincrement-Mode wird entsprechend verfahren, so daß die Register in der richtigen Reihenfolge vom Stack zurückgeholt werden.

Als Zielangabe sind für <ea> möglich:

(An)
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

Und als Quellangabe bei der zweiten Befehlsform gelten für <ea>:

(An)
(An)+
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)

MOVEP

Bedeutung: Move Peripheral Data
Transportiere Peripherie-Daten

Syntax: MOVEP.x Dx,d(Ay)
MOVEP.x d(Ay),Dx

Format: 0 0 0 0 dr dr dr op op op 0 0 1 ar ar ar

dr: Datenregister D0..D7
op: 100 Wort von Speicher nach Register
101 Langwort von Speicher nach Register
110 Wort von Register nach Speicher
111 Langwort von Register nach Speicher

Operand: Wort, Langwort

Flags: X N Z V C
\- - - - -
werden nicht verändert.

Dieser Befehl ist vorwiegend für 68xxx-Systeme mit 16-Bit breitem Datenbus gedacht, um 8-Bit-Peripherie einfacher zu bedienen, da sie dort auf nur jeweils geraden oder ungeraden Adressen zu liegen kommt.
Es kann eine Langwort oder Wortgröße transportiert werden. Beim Transport vom Datenregister zur Peripherie wird dann der Inhalt der Quelle in Bytes zerlegt und beginnend mit der höchstwertigen Stelle auf entweder geraden Adressen (Bit 15-Bit 8 des Datenbusses) oder

auf ungeraden Adressen (Bit 8-Bit 0 des Datenbusses) abgelegt, je nach dem ob die Zieladresse gerade oder ungerade ist.
Beispiel:

```
START:  MOVE.L #$12345678,D0      * Datenwert
        MOVEA.L #$FFFFFF00,A0     * Port Adresse
        MOVEP.L D0,0(A0)          * Transportieren
        RTS
```

Byte $12 wird auf Adresse $FFFFFF00 transportiert, Byte $34 auf Adresse $FFFFFF02, Byte $56 auf Adresse $FFFFFF04 und Byte $78 auf Adresse $FFFFFF06.

Beim Transport von der Peripherie in ein Register wird entsprechend umgekehrt verfahren.

MOVEQ

Bedeutung: Move Quick
Transportiere schnell

Syntax: MOVEQ #konst,Dn

Format: 0 1 1 1 r r r 0 b b b b b b b b

r: Register D0..D7
b: Byte-Konstante

Operand: Langwort

Flags:

X	N	Z	V	C
-	*	*	0	0

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V,C werden rückgesetzt.

X wird nicht verändert.

Als Ziel wird immer ein Langwort angesprochen. Die angegebene Konstante ist vorzeichenbehaftet und darf im Bereich -128 bis + 127 liegen. Der Befehl belegt nur ein Wort als Maschinencode und ist daher schneller als der ensprechende MOVE-Befehl.

Beispiel:

```
START:      MOVEQ #-10,D7
            RTS
```

Im Register D7 steht anschließend der Wert $FFFFFFF6, was -10 entspricht.

Manche Assembler verwenden diesen Befehl automatisch, wenn man z.B. MOVE #-10,D7 angibt, also eine Konstante im Bereich -128 bis +127 und ein Datenregister als Ziel.

MOVES *(nur 68010 und 68012)*

Bedeutung: Move Address Space
Transport im Adreßraum

Syntax: MOVES Rn,<ea>
MOVES <ea>,Rn

Format:

0	0	0	0	1	1	1	0	s	s	ea	ea	ea	ea	ea	ea
a/d	r	r	r	dr	0	0	0	0	0	0	0	0	0	0	0

s: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
ea: Adreßmode
a/d: 0=Datenregister
1=Adreßregister
r: Register 0..7
dr: 0= <ea> nach Register
1= Register nach <ea>

Operand: Byte, Wort, Langwort

Flags: X N Z V C
\- - - - -

werden nicht verändert

Dieser Befehl kann nur im System-Mode ausgeführt werden. Im User-Mode führt er zu einer Exception. Mit dem Befehl ist es möglich Daten zwischen unterschiedlichen Adreßräumen zu transportieren.

Dazu müssen die Register SFC oder DFC mit dem entsprechenden Funktionscode (siehe Kapitel 3) belegt werden.

Beispiel:

```
MOVE #2,D0                  * Code für User-Programm
MOVEC D0,SFC                * In Quell-Register
MOVE #1,D0                  * Code für User-Daten
MOVEC D0,DFC                * in Ziel-Register
MOVES.B $12345678,D0        * von User-Programm
MOVES.B D0,$456780123       * nach User-Daten
RTS
```

MULS

Bedeutung: Signed Multiply
vorzeichenbehaftete Multiplikation

Syntax: MULS <ea>,Dn

Operand: 1 1 0 0 r r r 1 1 1 ea ea ea ea ea ea

r: Register D0..D7
ea: Adreßmode

Operand: Wort

Flags:

X	N	Z	V	C
-	*	*	0	0

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V,C werden immer rückgesetzt.

Zwei 16-Bit-Datenwerte werden miteinander multipliziert und das 32-Bit Ergebnis abgelegt. Dabei wird der erste Operand mit dem zweiten multipliziert und das Ergebnis dort als 32-Bit Zahl abgelegt.

Beispiel:

```
START:      MOVE #-3,D0     * Wert 1
            MULS #5,D0      * mit Immediate multipl.
            RTS
```

Nach der Ausführung steht in Register D0.L das Ergebnis -15 oder i sedezimaler Schreibweise $FFFFFFF1.
Der MULS-Befehl ist auch zum DIVS kompatibel, so daß man z.I folgende Formel leicht auswerten kann:

ERG = (A * B + C * D + E * F) / G

Eine Formel, die in der Praxis z.B. bei 3D-Transformationen häufig vorkommt.
Beispiel:

```
START:     MOVE VARA,D0        * A * B ausführen
           MULS VARB,D0
           MOVE VARC,D1        * C * D
           MULS VARD,D1
           MOVE VARE,D2        * E * F
           MULS VARF,D2
           ADD.L D1,D0         * alles aufaddieren
           ADD.L D2,D0         * wichtig, Langworte nun
           DIVS VARG,D0        * Ergebnis in D0.W
           RTS

VARA:        DC.L xxxx  * hier Datenwerte ablegen
VARB:        DC.L xxxx
VARC:        DC.L xxxx
VARD:        DC.L xxxx
VARE:        DC.L xxxx
VARF:        DC.L xxxx
VARG:        DC.L xxxx
```

Als <ea> kann verwendet werden:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

MULU

Bedeutung Unsigned Multiply
vorzeichenlose Multiplikation

Syntax: MULU <ea>,Dn

Format: 1 1 0 0 r r r 0 1 1 ea ea ea ea ea ea

r: Register D0..D7
ea: Adreßmode

Operand: Wort

Flags: X N Z V C
- * * 0 0

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V,C werden immer rückgesetzt.

Zwei 16-Bit-Datenwerte werden miteinander multipliziert und ein 32-Bit Ergebnis abgelegt. Die Zahlen werden als vorzeichenlose Größen behandelt. Dabei wird der erste Operand mit dem zweiten multipliziert und das Ergebnis dort als 32-Bit Zahl abgelegt.

Beispiel:

```
START:    MOVE #$FF00,D0       * Wert 1
          MULU #5,D0           * mit Immediate multipl.
          RTS
```

Nach der. Ausführung steht im Register D0.L der Wert $0004FB00. Bei einem MULS-Befehl wäre dort der Wert $FFFFFB00 abgelegt worden. Hier wurde also die Zahl 65280 mit dem Wert 5 multipliziert. Das Ergebnis ist 326400 (sedezimal $4FB00).
Der MULU-Befehl ist auch zum DIVU kompatibel, so daß man z.B. folgende Formel leicht auswerten kann:

$$ERG = (A * B + C * D + E * F) / G$$

Eine Formel, die in der Praxis z.B. bei 3D-Transformationen häufiger vorkommt.

Beispiel:

```
START:    MOVE VARA,D0         * A * B ausführen
          MULU VARB,D0
          MOVE VARC,D1         * C * D
          MULU VARD,D1
          MOVE VARE,D2         * E * F
          MULU VARF,D2
          ADD.L D1,D0          * alles aufaddieren
          ADD.L D2,D0          * wichtig, Langworte nun
          DIVU VARG,D0         * Ergebnis in D0.W
          RTS
```

```
VARA:     DC.L xxxx  * hier Datenwerte ablegen
VARB:     DC.L xxxx
VARC:     DC.L xxxx
VARD:     DC.L xxxx
VARE:     DC.L xxxx
VARF:     DC.L xxxx
VARG:     DC.L xxxx
```

Achtung, hierbei werden die Variablen aber als vorzeichenlose Zahlen behandelt!

Als <ea> kann verwendet werden:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

NBCD

Bedeutung: Negate Decimal with Extend
Negiere dezimal mit Erweiterung

Syntax: NBCD <ea>

Format: 0 1 0 0 1 0 0 0 0 0 ea ea ea ea ea ea

ea: Adreßmode

Operand: Byte

Flags: X N Z V C
* ? * ? *

N undefiniert.

Z wird gelöscht, wenn das Ergebnis ungleich Null ist, sonst bleibt es unverändert. Achtung, wenn dasZ-Flag verwendet werden soll, so muß es vor dem ersten NBCD-Befehl gelöscht werden.

V undefiniert.

C,X werden gesetzt, wenn ein Übertrag ensteht.

Die angegebene Größe ist immer ein Byte-Wert. Die Negation wird in BCD-Arithmetik ausgeführt. Somit wird das Zehner komplement gebildet. Das X-Flag wird bei der Negation als Stellenübertrag

berücksichtigt, so daß der Befehl kaskadiert werden kann.

Beispiel:

```
START:    MOVE.B #$12,D0     * BCD-Zahl 12
          NBCD D0
          RTS
```

Anschließend steht in D0.B der Wert $88, was dem dezimalen Wert 88 in BCD-Darstellung entspricht. Die Zahl 88 erechnet sich aus 100 - 12.

Als <ea> kann man folgende Adressierarten verwenden:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

NEG

Bedeutung: Negate
Negiere

Syntax: NEG.x <ea>

Format: 0 1 0 0 0 1 0 0 size size ea ea ea ea ea ea

size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
ea: Adreßmode

Operand: Byte, Wort, Langwort

Flags:

X	N	Z	V	C
*	*	*	*	*

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V wird gesetzt, wenn ein Überlauf entsteht, sonst rückgesetzt.

C,X werden rückgesetzt, wenn das Ergebnis Null ist, sonst gesetzt!!

Der angegebene Operand wird von 0 subtrahiert und das Ergebnis wird dort wieder abgespeichert.

Beispiel:

```
MOVE.L #1,D0     * Wert laden
NEG.L D0         * Zweierkomplement bilden
RTS
```

In D0.L steht anschließend der Wert $FFFFFFFF, was dem Wert-1 in Zweierkomplementdarstellung entspricht.

Als <ea> kann verwendet werden:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

NEGX

Bedeutung: Negate with Extend
Negiere mit Erweiterung

Syntax: NEGX.x <ea>

Format: 0 1 0 0 0 0 0 0 0 0 size size ea ea ea ea ea ea

size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand

Operand: Byte, Wort, Langwort

Flags:

X	N	Z	V	C
*	*	*	*	*

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V wird gesetzt, wenn ein Überlauf entsteht, sonst rückgesetzt.

C,X werden gesetzt, wenn ein Übertrag entsteht, sonst rückgesetzt.

Der angegebene Operand und das X-Flag werden von 0 subtrahiert und das Ergebnis wird dort abgespeichert. Damit ist es möglich den Befehl zu kaskadieren.

Beispiel:

```
        MOVE #0,CCR       * Carry löschen zunächst
        NEGX.L ALPHA+4    * niederwertige Stelle zuerst
        NEGX.L ALPHA      * dann höherwertige
        RTS

ALPHA:  DC.L $00000000,$00000002   * Wert zwei Langworte
```

Danach steht in ALPHA der Wert $FFFFFFFF, $FFFFFFFE, was dem Wert -2 mit doppelter Langwortgenauigkeit entspricht.

Als <ea> sind möglich:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

NOP

Bedeutung: No Operation
Keine Operation

Syntax: NOP

Format: 0 1 0 0 1 1 1 0 0 1 1 1 0 0 0 1

Flags: X N Z V C
\- - - - -

werden nicht verändert.

Es wird keine Operation ausgeführt.Dieser Befehl wird gerne als Verzögerung verwendet, was allerdings nicht unproblematisch ist, da die 68xxx-Familie Befehle unterschiedlich schnell ausführt.

Beispiel:

```
          MOVE #1000,D1
SCHLEIFE: NOP                 * Verzögerung
          DBRA D1,SCHLEIFE    *
          RTS
```

Achtung: Die Ausführungszeiten auf 68008, 68000, 68010 und 68020 können dabei dramatisch variieren.

NOT

Bedeutung: Logical Complement
logische Komplementierung

Syntax: NOT.x <ea>

Format: 0 1 0 0 0 1 1 0 size size ea ea ea ea ea ea

size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
ea: Adreßmode

Operand: Byte, Wort, Langwort

Flags: X N Z V C
- * * 0 0

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V,C werden immer rückgesetzt.

Der Befehl bildet das Einer-Komplement. Das bedeutet, jede Binärstelle wird für sich allein und unabhängig komplementiert.

Beispiel:

```
MOVE.W #%0000001010111110,D2
NOT.W D2
RTS
```

In D2.W steht anschließend der Wert %1111110101000001.

Als <ea> sind erlaubt:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

OR

Bedeutung: Inclusive Or Logical
Einschließliche Oder-Verknüpfung

Syntax: OR.x <ea>,Dn
OR.x Dn,<ea>

Format: 1 0 0 0 r r r opm opm opm ea ea ea ea ea ea
r: Datenregister D0..D7
opm: 000 Byte-Operand <ea>,Dn
001 Wort-Operand <ea>,Dn
010 Langwort-Operand <ea>,Dn
100 Byte-Operand Dn,<ea>
101 Wort-Operand Dn,<ea>
110 Langwort-Operand Dn,<ea>

Operand: Byte, Wort, Langwort

Flags: X N Z V C
- * * 0 0

N wird gesetzt, wenn das höchstwertige Bit des Ergebnisses gesetzt ist, sonst wird es rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V,C werden immer rückgesetzt.

X bleibt unverändert.

Eine logische Oder-Verknüpfung wird zwischen den beiden Operanden durchgeführt und das Ergebnis im Ziel (zweiter Operand) abgespeichert.

Beispiel:

```
START:    MOVE #%0101,D0
          MOVE #%0011,D1
          OR D1,D0
          RTS
```

Anschließend steht in D0 der Wert %0111 oder dezimal 7.

Wenn <ea> als Quellangabe verwendet wird, sind folgende Adressierarten erlaubt:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

Als Zielangabe kann für <ea> verwendet werden:

(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

ORI

Bedeutung: Inclusive OR Immediate
Einschließliche Oder-Verknüpfung nachfolgend

Syntax: ORI.x #konst,<ea>

Format:

0 0 0 0 s s ea ea ea ea ea ea ea ea ea ea
w w w w w w w w b b b b b b b b
l l l l l l l l l l l l l l l l

s: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
ea: Adreßmode
b: Byte-Konstante
w: Wort-Konstante
l: Langwort-Konstante

Flags:

X N Z V C
- * * 0 0

N wird gesetzt, wenn das höchstwertige Bit des Ergebnisses gesetzt ist, sonst wird es rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V,C werden immer rückgesetzt.

X bleibt unverändert.

Eine logische Oder-Verknüpfung wird zwischen der angegebenen Konstanten und dem Zieloperand durchgeführt. Das Ergebnis wird dort abgespeichert.

Beispiel:

```
ORI.B #1,PORT
```

Bit 0 der Speicherzelle mit dem Namen PORT wird auf 1 gesetzt.

Als <ea> sind erlaubt:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

ORI to CCR

Bedeutung: Inclusive OR Immediate to Condition Code Register
Einschließliche Verknüpfung mit Bedingungscode-Register

Syntax: ORI #konst,CCR

Format: 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0
0 0 0 0 0 0 0 0 b b b b b b b b

b: Byte-Konstante

Operand: Byte

Flags: X N Z V C
* * * * *

X wird 1, wenn Bit 4 auf 1 ist.

N wird 1, wenn Bit 3 auf 1 ist.

Z wird 1, wenn Bit 2 auf 1 ist.

V wird 1, wenn Bit 1 auf 1 ist.

C wird 1, wenn Bit 0 auf 1 ist.

Damit lassen sich einzelne Flags setzen.

Beispiel:

```
SETCARRY:   ORI #1,CCR
            RTS

SETZERO:    ORI #4,CCR      * Binär %100.
            RTS
```

ORI to SR

Bedeutung: Inclusive OR Immediate to the Status Register
Einschließliche Oder-Verknüpfung mit dem Status-Register

Syntax: ORI #konst,SR

Format: 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0
w w w w w w w w w w w w w w w w

w: Wort-Konstante

Operand: Wort

Flags:

X	N	Z	V	C
*	*	*	*	*

Alle Flags werden verändert. Sie werden auf 1 gesetzt, wenn das entsprechende Bit in der Konstanten auf 1 liegt, sonst bleiben sie unverändert.
Dieser Befehl ist privilegiert und kann nur im System-Mode verwendet werden. Im User-Mode führt er zur einer Exception und damit zu einer Programmunterbrechung.

Die Belegung des Statusregisters lautet:

Bit 15	Trace-Mode
Bit 13	Supervisor-State
Bit 10 bis Bit 8	Interrupt-Level
Bit 4	X-Flag
Bit 3	N-Flag
Bit 2	Z-Flag
Bit 1	V-Flag
Bit 0	C-Flag

Beispiel:

```
ORI #$700,SR
```

Der Interrupt-Level wird auf 7 gesetzt und damit alle Interrupts gesperrt.

PEA

Bedeutung: Push Effective Address
Verwendete Adresse auf Stapel legen

Syntax: PEA <ea>

Format: 0 1 0 0 1 0 0 0 0 1 ea ea ea ea ea ea

ea: Adreßmode

Operand: Langwort

Flags: X N Z V C
- - - - -

werden nicht verändert.

Die Adresse des angegebenen Operanden wird auf den Stack (durch A7 verwaltet) gelegt.
Dabei wird immer ein Langwort verwendet.
Dieser Befehl kann z.B. zur Parameterübergabe an Unterprogramme verwendet werden.

Beispiel:

```
PEA 3(A0,D2.L)
```

Die so angegebene Adresse wird in Abhängigkeit von A0, D2.L ermittelt und auf den Stack gelegt. In diesem Fall ist es der Wert A0 + D2.L + 3.

Für <ea> können folgende Adressierarten verwendet werden:

(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)

RESET

Bedeutung: Reset External Devices
Externe Einheiten rücksetzen

Syntax: RESET

Format: 0 1 0 0 1 1 1 0 0 1 1 1 0 0 0 0

Flags: X N Z V C
\- - - - -

werden nicht verändert.

Die Reset-Leitung des Prozessors wird für kurze Zeit auf Low gelegt. Dieser Befehl ist privilegiert und darf nur im System-Mode verwendet werden. Im User-Mode führt er zu einer Exception und damit zur Programmunterbrechung.

Beispiel:

```
RESET        * reset all external devices
```

ROL, ROR

Bedeutung: Rotate (without Extend)
Rotiere (ohne Erweiterung)

Syntax:
ROL.x Dx,Dy
ROL.x #konst,Dy
ROL.x <ea>
ROR.x Dx,Dy
ROR.x #konst,Dy
ROR.x <ea>

Format: 1 1 1 0 c/r c/r c/r dr size size i/r 1 1 r r r **Register** schieben

c/r: wenn i/r=0, Konstante 0..7, die 8,1..7 Schiebeoperationen entspricht
wenn i/r=1, dann Register D0..D7

dr: =0, rechts schieben
=1, links schieben

size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand

i/r: =0, dann Konstante angegeben
=1, dann indirektes Schieben mit Register

1 1 1 0 0 1 1 dr 1 1 ea ea ea ea ea ea **Speicher** schieben

dr: =0, rechts schieben
=1, links schieben

ea: Adreßmode

Operand: Byte, Wort, Langwort

Flags: X N Z V C
- * * 0 *

N wird gesetzt, wenn das höchstwertige Bit des Ergebnisses den Wert 1 hat, sonst rückgesetzt.

Z wird gesetz bei Ergebnis Null , sonst rückgesetzt.

V wird immer rückgesetzt.

C entspricht dem letzten herausgeschobenen Bit, es wird jedoch immer rückgesetzt, wenn die Anzahl der Schiebeoperationen Null war.

Der angegebene Operand wird bitweise nach links, bzw. nach rechts verschoben. Bei einer Links-Schiebeoperation wird das höchstwertige Bit wieder an Bitposition 0 gesetzt. Bei einer Rechts-Schiebeoperation wird Bit 0 anschließend an die Stelle des höchstwertigen Bits gesetzt.

Die Anzahl der Schiebeoperationen kann bei Angabe einer Konstanten zwischen 1 und 8 liegen, bei der Angabe einer <ea> jedoch nur 1. Wird ein Register zusätzlich angegeben, so steht dort die Anzahl der Schiebeoperationen, die dann im Bereich 1 bis 32 liegen kann. Bei Angabe von <ea> kann nur eine Wortgröße angegeben werden.

Beispiel:

```
START:    MOVE.W #$8012,D0
          ROL.W #4,D0
          RTS
```

In D0.W steht danach der Wert $0128.

Als Adressierarten für <ea> sind erlaubt:

(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

ROXL, ROXR

Bedeutung: Rotate with Extent
Rotiere mit Erweiterung

Syntax:
ROXL.x Dx,Dy
ROXL.x #konst,Dy
ROXL.x <ea>
ROXR.x Dx,Dy
ROXR.x #konst,Dy
ROXR.x <ea>

Format:
1 1 1 0 c/r c/r c/r dr size size i/r 1 0 r r r **Register** schieben

c/r: wenn i/r=0, Konstante 0..7, die 8,1..7 Schiebeoperationen entspricht
wenn i/r=1, dann Register D0..D7
dr: =0, rechts schieben
=1, links schieben
size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
i/r: =0, dann Konstante angegeben
=1, dann indirektes Schieben mit Register

1 1 1 0 0 1 0 dr 1 1 ea ea ea ea ea ea **Speicher** schieben

dr: =0, rechts schieben
=1, links schieben
ea: Adreßmode

Operand: Byte, Wort, Langwort

Flags: X N Z V C
* * * 0 *

Flags:

N wird gesetzt, wenn das höchstwertige Bit des Ergebnisses auf 1 war, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V wird immer rückgesetzt.

C nimmt den Wert des zuletzt herausgeschobenen Bits an. Wird der Wert 0 als Anzahl der Schiebeoperationen angegeben, so wird C mit dem Inhalt von X geladen.

X nimmt den Wert des zuletzt herausgeschobenen Bits an. Wird der Wert 0 als Anzahl der Schiebeoperationen angegeben, so bleibt das Flag unverändert.

Der Operand wird je nach Befehl bitweise nach links oder rechts verschoben. Bei einem Links-Schiebebefehl wird das höchstwertige Bit in das X-Flag und C-Flag geschoben. Der alte Wert des X-Flags wird zuvor in die Position Bit 0 geschoben. Bei einem Rechts-Schiebebefehl wird das niederwertigste Bit in das X-Flag und C-Flag geschoben, der alte Wert des X-Flags wird zuvor an die Stelle des höchstwertigen Bits geschoben.

Beispiel:

```
            ASR ALPHA
            ROXR ALPHA+2
            ...
ALPHA:      DC.W $1234,$4567
```

Danach steht bei Alpha der Wert: $91A22B3.

Als <ea> sind erlaubt:

(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

RTD *(nur 68010 und 68012)*

Bedeutung: Return and Deallocate Parameters
Rückkehr und Platz für Parameter freigeben

Syntax: RTD #displacement

Format: 0 1 0 0 1 1 1 0 0 1 1 1 0 1 0 0
d d d d d d d d d d d d d d d d

d: Konstante 16-Bit

Operand: nicht bestimmt

Flags: X N Z V C
\- - - - -

werden nicht verändert.

Der Programmzähler wird vom Stack geholt, soweit verhält sich der Befehl wie RTS, dann wird die angegebene 16-Bit Konstante vorzeichenerweitert auf den Inhalt des Stackpointers addiert.

Beispiel:

```
UPRG:
....
RTD #10          * return und 10 Bytes freigeben
```

RTE

Bedeutung: Return from Exception
Rückkehr von dem Ausnahmezustand

Syntax: RTE

Format: 0 1 0 0 1 1 1 0 0 1 1 1 0 0 1 1

Operand: keiner

Flags: X N Z V C
* * * * *

werden gemäß des Stackinhalts gesetzt.

Dieser Befehl wird als Rücksprung aus einer Ausnahmebehandlung (exception) verwendet. Dazu gehört auch die Ausführung von TRAP-Befehlen. Der Befehl darf nur im System-Mode verwendet werden, der jedoch bei einer Ausnahmebehandlung automatisch eingestellt wird.
Statusregister und Programmzählerstand werden vom Stack zurückgeholt und das Programm setzt dann die Ausführung mit dem neuen Programmzählerstand fort. Normalerweise sind dies je zwei Langworte. Achtung: man sollte sich nicht auf den Inhalt des Stacks beziehen, da z.B. beim 68020 ggf. andere Werte dort mit abgelegt werden und Programme nicht mehr aufwärtskompatibel sind. Als Stack wird der System-Stack verwendet (A7-System).

Beispiel:

```
TRAPHANDLER:  ...    * Befehle zur Behandlung des
                       Ausnahmezustands
              RTE    * Ende, Hauptprogramm wieder
                       fortsetzen.
```

RTR

Bedeutung:	Return and Restore Condition Codes Kehre zurück und lade Bedingungsregister
Syntax:	RTR
Format:	0 1 0 0 1 1 1 0 0 1 1 1 0 1 1 1
Operand:	keiner
Flags:	X N Z V C * * * * *

Nur die Condition-Codes werden vom Stack zurück geladen, alle System-Bits im SR bleiben unverändert.

Zuerst werden die Condition-Codes als Wortgröße vom Stack geholt und danach die Rückkehradresse als Langwort in den Programmzähler.

Beispiel:

```
START:      BSR UNTERPRG      * aufrufen
            ...
UNTERPRG:   MOVE SR,-(A7)     * Status retten
            ...
            RTR               * Status zurück und Ende
```

Achtung: Der Befehl MOVE SR... ist bei 68010 und 68020 privilegiert und führt im User-Mode zu einer Exception. Diese Sequenz sollte daher vermieden werden, beim 68010, 68020 muß stattdessen der Befehl MOVE CCR,... eingesetzt werden, den es jedoch beim 68000 und 68008 nicht gibt.

RTS

Bedeutung: Return from Subroutine .
Kehre vom Unterprogramm zurück

Syntax: RTS

Format: 0 1 0 0 1 1 1 0 0 1 1 1 0 1 0 1

Operand: keiner

Flags: X N Z V C
\- - - - -

werden nicht verändert.

Rückkehr von einem Unterprogrammaufruf, der z.B. von BSR oder JSR durchgeführt wurde. Dazu wird ein Langwort vom Stack geholt und in den Programmzähler geladen. Der Befehl könnte der Operation "move.l (A7)+,programmzähler" entsprechen, den es aber so nicht gibt.

Beispiel:

```
START:    BSR UPRG      * aufrufen
          ...

UPRG:     ...
          RTS           * Rückkehr
```

SBCD

Bedeutung: Subtract Dezimal with Extend
Subtrahiere Dezimal mit Erweiterung

Syntax: SBCD Dy,Dx
SBCD -(Ay),-(Ax)

Format: 1 0 0 0 rx rx rx 1 0 0 0 0 r/m ry ry ry

rx: wenn r/m=0, dann Datenregister D0..D7
wenn r/m=1, dann Adreßregister A0..A7 für Predecrement

r/m: =0, dann Datenregister nach Datenregister
=1, dann indirekt über Adreßregister

ry: wenn r/m=0, dann Datenregister D0..D7
wenn r/m=1, dann Adreßregister A0..A7

Operand: Byte

Flags:

X	N	Z	V	C
*	?	*	?	*

N undefiniert.

Z wird rückgesetzt, wenn das Ergebnis Null war, sonst bleibt es unverändert. Achtung, daher sollte man das Flag zuvor löschen, wenn man es nach der Operation abfragen will.

V undefiniert.

C,X wird gesetzt, wenn ein dezimaler Übertrag entstanden ist, sonst wird es rückgesetzt.

Damit ist es möglich BCD-codierte Zahlen zu subtrahieren. Das X-Flag wird bei der Subtraktion entsprechend berücksichtigt, so daß der Befehl kaskadierbar ist. Der Befehl wird auf Byte-Größen angewendet, die zwei BCD-Zahlen in gepackter Form enthält. Siehe auch ABCD-Befehl.

Beispiel:

```
START:    MOVE.B #$99,D0     * erster Wert dezimal 99
          MOVE.B #$12,D1     * Zahl 12
          SBCD D1,D0         * D1-D0 nach D0
          RTS
```

Anschließend steht in D1.B der Wert $87, was in BCD-Schreibweise dem dezimalen Wert 87 entspricht.

SCC

Bedeutung: Set According to Condition Code
Setze, je nach Bedingungs-Code

Syntax: Scc <ea>
für cc siehe Bcc-Befehl

Format: 0 1 0 1 c c c c 1 1 ea ea ea ea ea ea

c: Condition-Code
ea: Adreßmode

Operand: Byte

Flags: X N Z V C
\- - - - -

werden nicht verändert.

Die angegebene Bedingung wird geprüft. Ist sie erfüllt, so wird der angegebene Operand mit Einsen gefüllt, sonst mit Nullen. Es kann nur eine Byte-Größe verwendet werden. Dieser Befehl läßt sich zum Setzen von Booleschen Variablen verwenden.

Beispiel:

```
CMP.W #1000,D0
SCC D3
```

D3.B erhält den Wert $FF, wenn der Inhalt von D0 gleich dem Wert 1000 war, sonst 0.

Als <ea> können verwendet werden:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

STOP

Bedeutung: Load Status Register and Stop
Lade Status-Register und halte

Syntax: STOP #konst

Format: 0 1 0 0 1 1 1 0 0 1 1 1 0 0 1 0
d d d d d d d d d d d d d d d d

d: Wort-Konstante

Operand: nicht bestimmt

Flags: X N Z V C
* * * * *

werden entsprechend der angegebenen Konstanten gesetzt, siehe ANDI #konst,SR.

X wird 0, wenn Bit 4 den Wert 0 hat, sonst unverändert.
N wird 0, wenn Bit 3 den Wert 0 hat, sonst unverändert.
Z wird 0, wenn Bit 2 den Wert 0 hat, sonst unverändert.
V wird 0, wenn Bit 1 den Wert 0 hat, sonst unverändert.
C wird 0, wenn Bit 0 den Wert 0 hat, sonst unverändert.

Die angegebene Konstante wird ins Status-Register geladen. Danach wird die Ausführung von Befehlen gestoppt. Der Programmzähler bleibt auf dem nachfolgenden Befehl stehen. Die Ausführung wird erst wieder nach einem Interrupt oder bei gesetztem Trace-Bit oder durch RESET fortgeführt. Der Befehl darf nur im System-Mode

ausgeführt werden, sonst erfolgt eine Exception.
Achtung: Das S-Bit muß in der Konstanten gesetzt sein, sonst erfolgt eine Exception.

Beispiel:

```
...
STOP #0
— hier immer privilege exception, da S-Bit = 0
...
```

SUB

Bedeutung: Subtract Binary
Subtrahiere Binär

Syntax: SUB.x <ea>,Dn
SUB.x Dn,<ea>

Format: 1 0 0 1 r r r opm opm opm ea ea ea ea ea ea

r: Register D0..D7
opm: 000 Byte-Operand <ea>,Dn
001 Wort-Operand <ea>,Dn
010 Langwort-Operand <ea>,Dn
100 Byte-Operand Dn,<ea>
101 Wort-Operand Dn,<ea>
110 Langwort-Operand Dn,<ea>
ea: Adreßmode

Operand: Byte, Wort, Langwort

Flags: X N Z V C
* * * * *

N wird gesetzt bei Ergebnis negativ, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V wird gesetzt bei Überlauf, sonst rückgesetzt.

C,X wird gesetzt, wenn ein Übertrag erzeugt wurde, sonst rückgesetzt.

Subtraktion zweier Zahlen. Der erste Operand wird vom zweiten subtrahiert und dort das Ergebnis abgespeichert. Dabei kann eine Byte-, Wort- oder Langwort-Größe verwendet werden.

Beispiel:

```
START:      LEA ALPHA,A0
            SUB #2,(A0)
            RTS

ALPHA:      DC.W $FFF
```

Anschließend steht in ALPHA der Wert $FFD.

Als Quellangabe sind für <ea> erlaubt:

Dn
An (nur bei Wort - oder Langwort-Subtraktion)
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

Als Zielangabe kann für <ea> gewählt werden:

(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

SUBA

Bedeutung: Subtract Address
Subtrahiere Adresse

Syntax: SUBA.W <ea>,An
SUBA.L <ea>,An

Format: 1 0 0 1 r r r opm opm opm ea ea ea ea ea ea

r: Register A0..A7
opm: 011 Wort-Operand
111 Langwort-Operand
ea: Adreßmode

Operand: Wort, Langwort

Flags: X N Z V C
\- - - - -

werden nicht verändert.

Der erste Operand wird vom Inhalt des angegebenen Adreßregisters subtrahiert und das Ergebnis im Adreßregister abgelegt. Wird als Quelle eine Wortgröße angegeben, so wird diese vor der Subtraktion zum Langwort erweitert (vorzeichenbehaftet) und als Ergebnis ein Langwort abgelegt.

Beispiel:

```
START:     MOVEA.L #$100000,A0     * Beispiel Wert
           SUBA.L #$FFFF,A0
           RTS
```

Anschließend steht im Register A0.L der Wert $100001, da $FFFF als vorzeichenbehaftete Zahl interpretiert wurde und damit die Operation $1000000 - (-1) ausgeführt wurde.

Als <ea> kann verwendet werden:

Dn
An
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L
d(PC)
d(PC,Xi)
#Imm

SUBI

Bedeutung: Subtract Immediate
Subtrahiere nachfolgend

Syntax: SUBI.x #konst,<ea>

Format:

0	0	0	0	0	1	0	0	s	s	ea	ea	ea	ea	ea	ea
w	w	w	w	w	w	w	w	b	b	b	b	b	b	b	b
l	l	l	l	l	l	l	l	l	l	l	l	l	l	l	l

size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
ea: Adreßmode
b: Byte-Konstante
w: Wort-Konstante
l: Langwort-Konstante

Operand: Byte, Wort, Langwort

Flags: X N Z V C
* * * * *

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V wird gesetzt bei Überlauf, sonst rückgesetzt.

C,X wird gesetzt bei Übertrag, sonst rückgesetzt.

Subtraktion einer Konstanten von einem Wert. Dabei kann hier im Gegensatz zum normalen SUB-Befehl auch eine Speicherzelle als Ziel angegeben werden.

Beispiel:

```
START;      LEA BUFFER,A3       * Adresse Buffer
            SUBI #10,(A3)
            RTS

BUFFER:     DC.W 15
```

In Buffer steht anschließend der Wert 5.

Als <ea> kann angegeben werden:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

SUBQ

Bedeutung: Subtract Quick
Subtrahiere schnell

Syntax: SUBQ.x #konst,<ea>

Format: 0 1 0 1 d d d 1 size size ea ea ea ea ea ea

d: Konstante, 0,1..7 entspricht 1, 2....8 als Datenwert
size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
ea: Adreßmode

Flags: X N Z V C
* * * * *

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn das Ergebnis Null ist, sonst rückgesetzt.

V wird gesetzt bei Überlauf, sonst rückgesetzt.

C,X wird gesetzt bei Übertrag, sonst rückgesetzt.

Konstanten im Bereich 1..8 können mit diesem Befehl von einem Wert subtrahiert werden. Dabei belegt der Maschinencode des Befehls nur ein Wort und ist damit in der Ausführung sehr schnell.

Bei Adreßregistern als Ziel wird immer nur mit einem Langwort gerechnet.

Beispiel:

```
START:     MOVE.B #100,D7   * Startwert laden
SCHLEIFE: ...               * diverse Befehle
           SUBQ.B #3,D7     * Zähle in 3er Schritten
           BGE ZAEHLE       * größer gleich Null, wiederhole
           RTS              * Sonst Ende
```

Die Schleife wird solange durchlaufen, solange der Inhalt von D7.B größer oder gleich Null ist.

Für <ea> kann stehen:

Dn
An (nur bei Wort - oder Langwort-Subtraktionen)
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

SUBX

Bedeutung: Subtract with Extend
Subtrahiere mit Erweiterung

Syntax: SUBX.x Dy,Dx
SUBX.x -(Ay),-(Ax)

Format: 1 0 0 1 rx rx rx 1 size size 0 0 r/m ry ry ry

rx: wenn r/m=0, dann Datenregister D0..D7
wenn r/m=1, dann Adreßregister A0..A7 für predecrement
size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
ry: wenn r/m=0, dann Datenregister D0..D7
wenn r/m=1, dann Adreßregister A0..A7 für predecrement

Operand: Byte, Wort, Langwort

Flags: X N Z V C
* * * * *

N wird gesetzt, wenn das Ergebnis negativ ist, sonst rückgesetzt.

Z wird rückgesetzt, wenn das Ergebnis ungleich Null ist, sonst bleibt es unverändert. Soll das Flag anschließend verwendet werden, so muß es vorher auf 1 gesetzt werden.

V wird gesetzt, wenn ein Überlauf stattfindet, sonst rückgesetzt.

C,X wird bei einem Übertrag gesetzt, sonst rückgesetzt.

Mit diesem Befehl kann man mehrere Subtraktionen nacheinander ausführen um die Wortlänge zu erhöhen.

Beispiel:

```
START:  LEA ZAHL1,A0          * erste Zahl, Adresse
        LEA ZAHL2,A1          * zweite Zahl, Adresse
        MOVE #$4,CCR          * X-Flag löschen, Z-Flag setzen
        SUBX.L -(A0),-(A1)    * niederwertigste Stelle zuerst
        SUBX.L -(A0),-(A1)    *
        SUBX.L -(A0),-(A1)    *
        SUBX.L -(A0),-(A1)    * höchstwertigste Stelle zuletzt
        RTS

        DC.L x1
        DC.L x2
        DC.L x3
        DC.L x4
ZAHL1:  DS 0
        DC.L y1
        DC.L y2
        DC.L y3
        DC.L y4
ZAHL2:  DS 0
```

Der Inhalt von ZAHL1 (x1..x4) wird von ZAHL2 (y1..y4) subtrahiert und das Ergebnis wird dort abgelegt.

SWAP

Bedeutung: Swap Register Halfes
tausche Registerhälften

Syntax: SWAP Dn

Format: 0 1 0 0 1 0 0 0 0 1 0 0 0 r r r

r: Register D0..D7

Operand: Wort

Flags: X N Z V C
- * * 0 0

N wird gesetzt, wenn das höchstwertige Bit des 32-Bit-Ergebnisses gesetzt ist, sonst rückgesetzt.

Z wird gesetzt, wenn das 32-Bit-Ergebnis 0 ist, sonst rückgesetzt.

C,V werden immer rückgesetzt.

Die beiden 16-Bit-Hälften eines Registers werden miteinander vertauscht.

Beispiel:

```
START:    MOVE.L #$12345678,D0
          SWAP D0
          RTS
```

Danach steht in Register D0.L der Wert $56781234.
Sollen die beiden 8-Bit-Hälften eines Registers vertauscht werden, so muß man einen ROR- oder ROL-Befehl verwenden.

Beispiel:

```
          MOVE.W #$1234,D0
          ROR #8,D0
```

Danach steht der Wert $3412 in Register D0.W.

TAS

Bedeutung: Test and Set an Operand
Prüfe und Setze Operand

Syntax: TAS <ea>

Format: 0 1 0 0 1 0 1 0 1 1 ea ea ea ea ea ea

ea: Adreßmode

Operand: Byte

Flags: X N Z V C
- * * 0 0

N wird gesetzt, wenn das höchstwertige Bit des Ausgangswertes gesetzt ist, sonst rückgesetzt.

Z wird gesetzt, wenn der Ausgangswert Null ist, sonst rückgesetzt.

C,V werden immer rückgesetzt.

Der aktuelle Wert der angegebenen Größe (immer nur ein Byte-Wert), wird geprüft und das N und Z-Flag entsprechend gesetzt. Anschließend wird das höchstwertige Bit des Operanden auf 1 gesetzt. Diese Operation wird mit einem RMW-Zyklus durchgeführt, der nicht unterbrechbar ist. Damit eignet sich die Operation speziell für Multiprozessor-Anwendungen, da sie unteilbar ist, somit kann ein

anderer Prozessor die Speicherzelle nicht vor Abschluß des Befehls auslesen oder verändern.

Beispiel:

```
        TAS ALPHA
        ...
ALPHA:  DC.B 0
```

Anschließend steht in ALPHA der Wert $80. Das Z-Flag ist gesetzt und N ist rückgesetzt.

Als <ea> kann stehen:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

TRAP

Bedeutung: Trap
Systemsprung (Falle)

Syntax: TRAP #konst

Format: 0 1 0 0 1 1 1 0 0 1 0 0 v v v v

v: Vektornummer 0..15

Operand: nicht bestimmt

Flags: X N Z V C
\- - - - -

werden nicht verändert.

Durch den Befehl wird die sogenannte Trap-Exception veranlaßt. Dadurch wird die Programmausführung unterbrochen und das TRAP-Unterprogramm im System-Mode aufgerufen. Die Konstante, die im Bereich 0..15 liegen darf, bestimmt welcher Trap-Vektor verwendet werden darf.

Beispiel:

```
TRAP #1
...
```

TRAPV

Bedeutung: Trap on Overflow
Systemaufruf wenn Überlauf

Syntax: TRAPV

Format: 0 1 0 0 1 1 1 0 0 1 1 1 0 1 1 0

Operand: keiner

Flags: X N Z V C
- - - - -

werden nicht verändert.

Wenn das V-Flag gesetzt ist, wird eine TRAPV-Exception durchgeführt und die normale Programmausführung unterbrochen, sonst verhält sich der Befehl wie NOP.
Damit ist es möglich einen arithmetischen Überlauf zu erkennen und eine Fehlerbehandlung einzuleiten.

Beispiel:

```
...             * arithmetische Befehle
TRAPV           * Exeception wenn Überlauf
...
```

Bei einem arithmetischen Überlauf (V-Flag gesetzt) wird die Exception ausgelöst und die Programmausführung unterbrochen.

TST

Bedeutung: Test an Operand
Prüfe Operanden

Syntax: TST.x <ea>

Format: 0 1 0 0 1 0 1 0 size size ea ea ea ea ea ea

size: 00 Byte-Operand
01 Wort-Operand
10 Langwort-Operand
ea: Adreßmode

Operand: Byte, Wort, Langwort

Flags:

X	N	Z	V	C
-	*	*	0	0

N wird gesetzt, wenn der Operand Negativ ist, sonst rückgesetzt.

Z wird gesetzt, wenn der Operand Null ist, sonst rückgesetzt.

V,C werden immer rückgesetzt.

Der angegebene Operand wird mit der Zahl 0 verglichen und die Flags werden entsprechend gesetzt. Der Operand wird nicht verändert.

Beispiel:

```
         TST.L ALPHA
         ...
ALPHA:   DC.L $80000000
```

Das N-Flag ist anschließend gesetzt und das Z-Flag rückgesetzt.

Für <ea> kann stehen:

Dn
(An)
(An)+
-(An)
d(An)
d(An,Xi)
Abs.W
Abs.L

UNLK

Bedeutung: Unlink
Binden rückgängig machen

Syntax: UNLK An

Format: 0 1 0 0 1 1 1 0 0 1 0 1 1 r r r

r: Adreßregister A0..A7

Operand: nicht bestimmt

Flags: X N Z V C
\- - - - -

werden nicht verändert.

Dieser Befehl bildet das Gegenstück zum LINK-Befehl (siehe LINK-Befehl). Der Stackpointer wird mit dem Inhalt des angegebenen Adreßregisters geladen, dann wird anschließend das Adreßregister mit einem neuen Langwort geladen, das dazu zuvor vom Stack (neuer Stackpointerwert) geholt wird.

Beispiel:

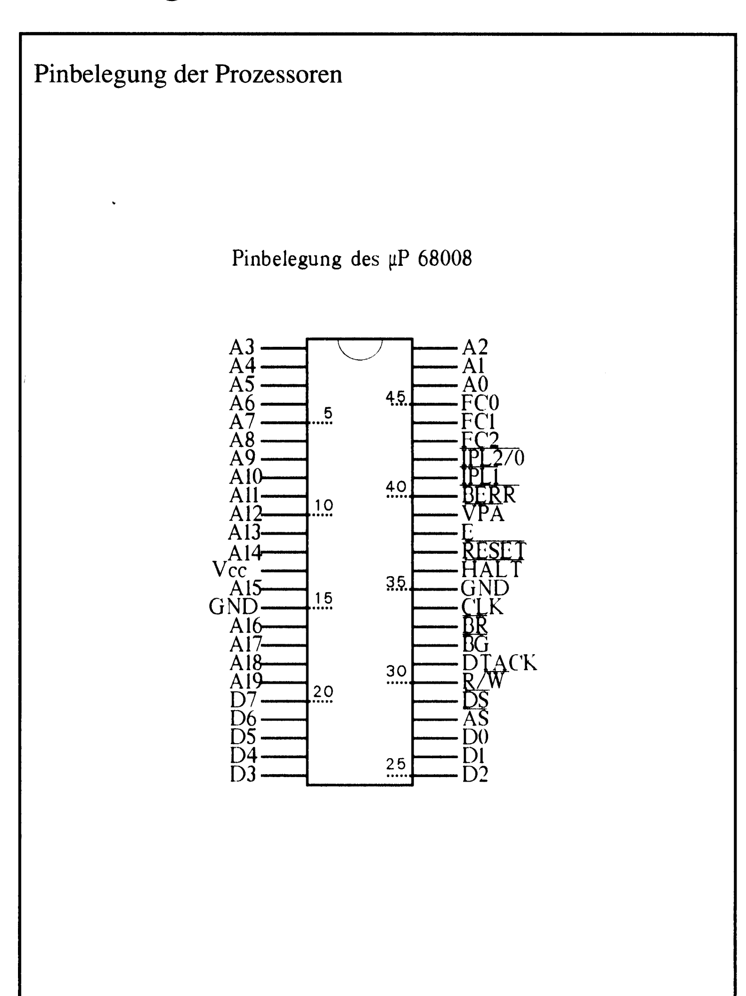
```
UPRG:   LINK A6,#-10    * Platz reservieren
        ...             * div. Befehle
        UNLK A6         * wieder rückgängig machen
        RTS
```

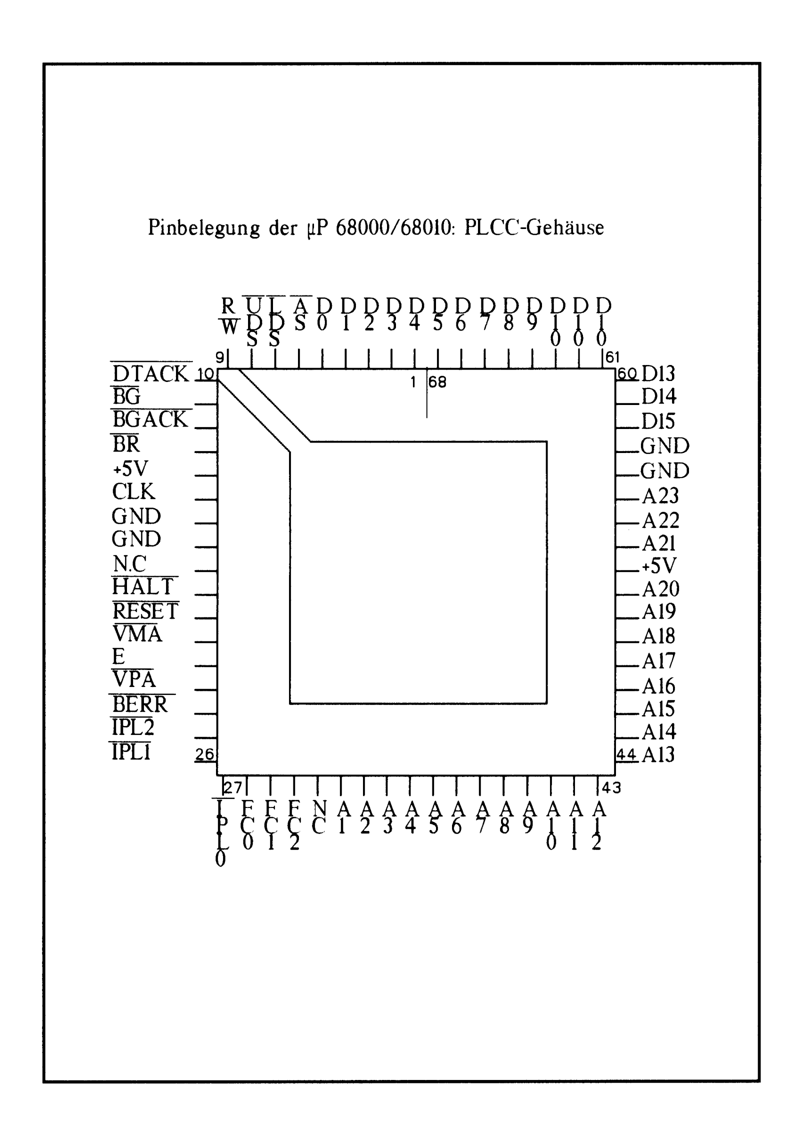
Es werden 10 Speicherzellen auf dem Stack reserviert. Dort lassen sich z.B. lokale Daten unterbringen. Achtung, die Reservierung erfolgt durch Angabe einer negativen Zahl, da der Stack immer abfallende Adressen belegt. Der Zugriff auf diese lokalen Daten kann dann z.B. mit Adressierarten wie -10(A6),-2(A6) ... erfolgen.



Anhang A

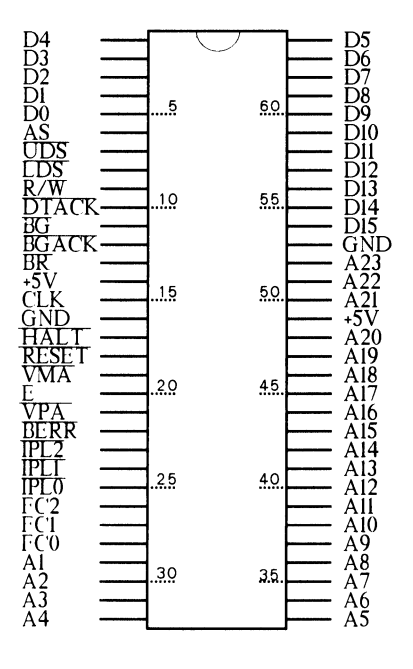
Pinbelegung der Prozessoren



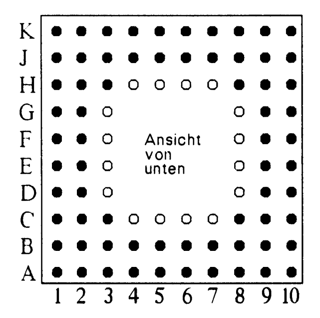


Pinbelegung der µP 68000/68010: PLCC-Gehäuse

Pinbelegungen der µP 68000/68010: DIL-Gehäuse



Pinbelegung der µP 68000/68010/68012: Pin-Grid-Gehäuse



Pinbelegung: ● 68000/68010/68012, ○ ***nur 68012***

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
A1	NC	C1	$\overline{\text{BGACK}}$	G1	$\overline{\text{VMA}}$	K1	NC
A2	$\overline{\text{AS}}$	C2	$\overline{\text{BG}}$	G2	$\overline{\text{VPA}}$	K2	FC2
A3	D1	C3	R/$\overline{\text{W}}$	***G3***	***A27***	K3	FC0
A4	D2	***C4***	***$\overline{\text{RMC}}$***	G9	A15	K4	A1
A5	D4	C8	D13	G10	A17	K5	A3
A6	D5	C9	A23	H1	E	K6	A4
A7	D7	C10	A22	H2	$\overline{\text{IPL2}}$	K7	A6
A8	D8	D1	$\overline{\text{BR}}$	H3	$\overline{\text{IPL1}}$	K8	A7
A9	D10	D2	+5V	***H5***	***A29***	K9	A9
A10	D12	***D8***	***GND***	***H6***	***A28***	K10	NC
B1	$\overline{\text{DTACK}}$	D9	GND	H8	A13		
B2	$\overline{\text{LDS}}$	D10	A21	H9	A12		
B3	$\overline{\text{UDS}}$	E1	CLK	H10	A16		
B4	D0	E2	GND	J1	$\overline{\text{BERR}}$		
B5	D3	***E3***	***GND***	J2	$\overline{\text{IPL0}}$		
B6	D6	***E8***	***A24***	J3	FC1		
B7	D9	E9	+5V	***J4***	***A31***		
B8	D11	E10	A20	J5	A2		
B9	D13	F1	$\overline{\text{HALT}}$	J6	A5		
B10	D15	F2	$\overline{\text{RESET}}$	J7	A8		
		F8	***A25***	J8	A10		
		F9	A18	J9	A11		
		F10	A19	J10	A14		

Pin J4: NC bei 68000, A31 bei 68012

Anhang B

Kurzreferenz der Befehle

Befehl	X	N	Z	V	C	Beschreibung
ABCD.B Dx,Dy	X	N	Z	V	C	Addition von BCD-Zahlen
ABCD.B -(An),-(Am)	X	N	Z	V	C	Addition von BCD-Zahlen
ADD.x <ea>,Dn	X	N	Z	V	C	Binäre Addition
ADD.x Dn,<ea>	X	N	Z	V	C	Binäre Addition
ADDA.x <ea>,An	-					Binäre Addition mit Adreßregister als Ziel
ADDI.x #konst,<ea>	X	N	Z	V	C	Binäre Addition mit Konstante ale Quellop.
ADDQ.x #konst,<ea>	X	N	Z	V	C	Addiere schnell (Konstante 1..8)
ADDX.x Dx,Dy	X	N	Z	V	C	Addiere mit Extend-Flag
ADDX.x -(Ax),-(Ay)	X	N	Z	V	C	Addiere mit Extend-Flag
AND.x <ea>,Dn	-	N	Z	V	C	Logische Und-Verknüpfung
AND.x Dn,<ea>	-	N	Z	V	C	Logische Und-Verknüpfung
ANDI.x #konst,<ea>	-	N	Z	V	C	Logische Und-Verkn. mit einer Konstanten
ANDI #konst,CCR	X	N	Z	V	C	Und-Verknüpfung mit Bedingungscode-Register
ANDI #konst,SR	X	N	Z	V	C	(*) Und-Verknüpfung mit Status-Register
ASL.x Dx,Dy	X	N	Z	V	C	Links Schieben indirekt
ASL.x #konst,Dn	X	N	Z	V	C	Links Schieben um “konst” Stellen
ASL <ea>	X	N	Z	V	C	Links Schieben um eine Stelle
ASR.x Dx,Dy	X	N	Z	V	C	Rechts Schieben indirekt
ASR.x #konst,Dn	X	N	Z	V	C	Rechts Schieben um “konst” Stellen
ASR <ea>	X	N	Z	V	C	Rechts Schieben um eine Stelle
Bcc marke	-					Bedingter Sprung, 16-Bit Adreßdistanz
Bcc.S marke	-					Bedingter Sprung, 8-Bit-Adreßdistanz
BCHG Dx,<ea>	-	-	Z	-	-	Prüfe ein Bit und komplementiere es

Befehl	X	N	Z	V	C	Beschreibung
BCHG #konst,<ea>	-	-	Z	-	-	Prüfe ein Bit und komplementiere es
BCLR Dx,<ea>	-	-	Z	-	-	Prüfe ein Bit und setze es auf Null
BCLR #konst,<ea>	-	-	Z	-	-	Prüfe ein Bit und setze es auf Null
BKPT #konst	-					(+) Breakpoint
BRA marke	-					unbedingter Sprung, 16-Bit Adreßdistanz
BRA.S marke	-					unbedingter Sprung, 16-Bit Adreßdistanz
BSET Dn,<ea>	-	-	Z	-	-	Prüfe ein Bit und setze es auf Eins
BSET #konst,<ea>	-	-	Z	-	-	Prüfe ein Bit und setze es auf Eins
BSR marke	-					Unterprogrammaufruf, 16-Bit Adreßdistanz
BSR.S marke	-					Unterprogrammaufruf, 8-Bit Adreßdistanz
BTST Dn,<ea>	-	-	Z	-	-	Prüfe ein Bit
BTST #konst,<ea>	-	-	Z	-	-	Prüfe ein Bit
CHK <ea>,Dn	-	N	Z	V	C	Prüfe gegen Grenzbereich
CLR.x <ea>	-	N	Z	V	C	Setze Ziel auf 0
CMP.x <ea>,Dn	-	N	Z	V	C	Vergleich zweier Operanden
CMPA.x <ea>,An	-	N	Z	V	C	Vergleich mit einem Adreßregister
CMPI.x #konst,<ea>	-	N	Z	V	C	Vergleich einer Konstanten mit einemOperanden
CMPM.x (Ax)+,(Ay)+	-	N	Z	V	C	Vergleich zweier Operanden
DBcc Dn,marke	-					Prüfe Bedingung, decremen tiere und springe bedingt
DBRA Dn,marke	-					Decrementiere und springe bedingt
DIVS <ea>,Dn	-	N	Z	V	C	Division mit Vorzeichen

Befehl	X	N	Z	V	C	Beschreibung
DISU <ea>,Dn	-	N	Z	V	C	Division ohne Vorzeichen
EOR.x Dn,<ea>	-	N	Z	V	C	Logische Exklusiv-OderVerkn.
EORI.x #konst,<ea>	-	N	Z	V	C	Logische Exklusiv-Oder-Verknüpfung
EORI #konst,CCR	X	N	Z	V	C	Exklusiv-Oder mit Bedingscode-Register
EORI #konst,SR	X	N	Z	V	C	(*)Exklusiv-Oder mit Status-Register
EXG Rx,Ry	-					Vertausche Registerinhalte
EXT.x Dn	-	N	Z	V	C	Vorzeichenrichtige Erweiterung
ILLEGAL	-					Unzuläßiger Befehl, führt zur Exception
JMP <ea>	-					Springe Absolut, 32-Bit-Absolutadresse
JSR <ea>	-					Unterprogrammaufruf, 32-Bit-Absolutadresse
LEA <ea>,An	-					Lade effektive Adresse in Adreßregister
LINK An,#konst	-					Rette A7, und lege Stackbereich an
LSL.x Dx,Dy	X	N	Z	V	C	Logische Verschiebung nach Links, indirekt
LSL.x #konst	X	N	Z	V	C	Logische Verschiebung nach Links
LSL.x <ea>	X	N	Z	V	C	Logische Verschiebung nach Links, um Eins
LSR.x Dx,Dy	X	N	Z	V	C	Logische Verschiebung nach Rechts, indirekt
LSR.x #konst	X	N	Z	V	C	Logische Verschiebung nach Rechts
LSR.x <ea>	X	N	Z	V	C	Logische Verschiebung nach Rechts, um Eins
MOVE.x <ea>,<ea>	-	N	Z	V	C	Übertrage Daten von Quelle nach Ziel

MOVE <ea>,CCR	X	N	Z	V	C	Übertrage Wert nach Bedingungscode-Register
MOVE CCR,<ea>	-					(+) Übertrage Bedingungscode-Register nach Ziel
MOVE <ea>,SR	X	N	Z	V	C	(*) Übertrage Wert nach Status-Register
MOVE SR,<ea>	-					(* bei 68010/12) Übertrage Wert von Status-Register nach Ziel
MOVE USP,An	-					(*) Anwenderstackpointer nach Adreßregister
MOVE An,USP	-					(*) Adreßregister nach Anwenderstackpointer
MOVEA.x <ea>,An	-					Übertrage Wert nach Adreßregister
MOVEC Rc,Rn	-					(+,*) Übertrage Steuer-Register
MOVEC Rn,Rc	-					(+,*) Übertrage Steuer-Register
MOVEM.x reglist,<ea>	-					Mehrfach Register nach Ziel
MOVEM.x <ea>,reglist	-					Mehrfach Quelle nach Register
MOVEP.x Dx,d(Ay)	-					Peripherietransport
MOVEP.x d(Ax),Dy	-					Peripherietransport
MOVEQ #konst,Dn	-					Laden einer Konstanten, schnell
MOVES Rn,<ea>	-					(+,*) Übertrage in Adreßraum
MOVES <ea>,Rn	-					(+,*) Übertrage in Adreßraum
MULS <ea>,Dn	X	N	Z	V	C	Multiplikation mit Vorzeichen
MULU <ea>,Dn	X	N	Z	V	C	Multiplikation ohne Vorzeichen
NBCD <ea>	X	N	Z	V	C	Negation von BCD-Zahlen
NEG.x <ea>	X	N	Z	V	C	Negation, Zweierkomplement
NEGX.x <ea>	X	N	Z	V	C	Negation mit Extend-Flag
NOP	-					Keine Operation
NOT.x <ea>	X	N	Z	V	C	Logisches Einerkomplement

Befehl	X	N	Z	V	C	Beschreibung
OR.x <ea>,Dn	-	N	Z	V	C	Logische Oder-Verknüpfung
OR.x Dn,<ea>	-	N	Z	V	C	Logische Oder-Verknüpfung
ORI.x #konst,<ea>	-	N	Z	V	C	Logische Oder-Verknüpfung mit Konstantem
ORI #konst,CCR	X	N	Z	V	C	Logische Oder-Verknüpfung mit Bedingungscode-Register
ORI #konst,SR	X	N	Z	V	C	(*) Logische Oder-Verknüpfung mit Status-Register
PEA <ea>	-					Lege effektive Adresse auf Stack ab.
RESET	-					(*) Rücksetzimpuls auf RESET-Leitung auslösten
ROL.x Dx,Dy	X	N	Z	V	C	Rotiere Links, indirekt
ROL.x #konst,Dy	X	N	Z	V	C	Rotiere Links, um Anzahl "konst" Bits
ROL.x <ea>	X	N	Z	V	C	Rotiere Links, um Eins
ROR.x Dx,Dy	X	N	Z	V	C	Rotiere Rechts, indirekt
ROR.x #konst,Dy	X	N	Z	V	C	Rotiere Rechts, um Anzahl "konst" Bits
ROR.x <ea>	X	N	Z	V	C	Rotiere Rechts, um Eins
ROXL.x Dx,Dy	X	N	Z	V	C	Rotiere Links, indirekt mit X-Flag
ROXL.x #konst,Dy	X	N	Z	V	C	Rotiere Links, um Anzahl "konst" Bits mit X-Flag
ROXL.x <ea>	X	N	Z	V	C	Rotiere Links, um Eins mit X-Flag
ROXR.x Dx,Dy	X	N	Z	V	C	Rotiere Rechts, indirekt mit X-Flag
ROXR.x #konst,Dy	X	N	Z	V	C	Rotiere Rechts, um Anzahl "konst" Bits mit X-Flag
ROXR.x <ea>	X	N	Z	V	C	Rotiere Rechts, um Eins mit X-Flag
RTD	-					(+) Rückkehr und Parameter freigeben

Befehl	X	N	Z	V	C	Beschreibung
RTE	X	N	Z	V	C	(*) Rückkehr aus Exception
RTR	X	N	Z	V	C	Rückkehr mit Laden von Flags
RTS	-					Rückkehr aus Unterprogramm
SBCD Dx,Dy	X	N	Z	V	C	Subtraktion von BCD-Zahlen
SBCD -(Ax),-(Ay)	X	N	Z	V	C	Subtraktion von BCD-Zahlen
Scc	X	N	Z	V	C	Setze ein Byte je nach Bedingung
STOP #konst	X	N	Z	V	C	Lade Status-Register und HALT
SUB.x <ea>,Dn	X	N	Z	V	C	Binäre Subtraktion
SUBx Dn,<ea>	X	N	Z	V	C	Binäre Subtraktion
SUBA.x <ea>,An	-					Binäre Subtraktion mit Ziel Adreßregister
SUBI.x #konst,<ea>	X	N	Z	V	C	Binäre Subtraktion mit einer Konstanten
SUBQ.x #konst,<ea>	X	N	Z	V	C	Binäre Subtraktion schnell
SUBX.x Dx,Dy	X	N	Z	V	C	Binäre Subtraktion mit Extend-Flag
SUBX.x -(Ax),-(Ay)	X	N	Z	V	C	Binäre Subtraktion mit Extend-Flag
SWAP Dn	-	N	Z	V	C	Tausche Inhalt von Registerhälften
TAS <ea>	-	N	Z	V	C	Prüfe und Setze ein Bit im Zieloperand
TRAP #konst	-					Trap-Aufruf (Trap-Exception)
TRAPV	-					Trap bei Überlauf
TST.x <ea>	-	N	Z	V	C	Prüfe den Inhalt eines Operanden
UNLK An	-					Korrigiere Stack und A7 zurück

* = privilegierter Befehl, führt im User-Mode zur Exception
+ = Befehl nur beim 68010/68012 vorhanden

Anhang C

Kurzübersicht der Adressierungsarten

Register direkt

Datenregister direkt

clr D0

Adressregister direkt

movea A1,A6

Statusregister direkt

move D0,SR

CCR-Register direkt

move D0,CCR

Daten unmittelbar

unmittelbar lang

move #$20000,D0

unmittelbar kurz

move #$20,D0

unmittelbar schnell

moveq #9,D0

unmittelbar Zahlen 1-8

addq #1,D0

absolut

absolut kurz

not $2000

absolut lang

neg $14000

Register indirekt

Adressregister indirekt mit Postinkrement

clr (A4)+

Adressregister indirekt

clr (A0)

Adressregister indirekt Adressdistanz und Index

move 4(A0,D0),$1000

Adressregister indirekt mit Adressdistanz

clr $100(A0)

Adressregister indirekt mit Predekrement

clr -(A3)

relativ zum Programmzähler

PC-relativ mit Adressdistanz und Index

move $20(PC,A0),D0

PC-relativ mit Adressdistanz

move $20(PC),A0

PC-relativer Branch-Befehl

bne Marke

Anhang D

Ausführungszeiten der 68000-Befehle

Datenbewegung		Flags					Datenlänge	Ausführungszeit (Taktzyklen)												
		x	n	z	v	c		-	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rx)	xxxx.W	xxxxxx.L	d(PC)	d(PC,Rx)	#Op
move .	(ea),(ea)	-	×	×	0	0	8,16 von	-	2	2	6	6	8	10	12	10	14	10	12	6
							8,16 nach	-	+2	-	+6	+6	+6	+10	+12	+10	+14	-	-	-
							32 von	-	2	2	10	10	12	14	16	14	18	14	16	10
							32 nach	-	+2	-	+10	+10	+10	+14	+16	+14	+18	-	-	-
movea	(ea),An	-	-	-	-	-	16	-	4	4	8	8	10	12	14	12	16	12	14	8
							32	-	4	4	12	12	14	16	18	16	20	16	18	12
moveq	#Op,Dn	-	×	×	0	0	32	4	-	-	-	-	-	-	-	-	-	-	-	-
movem	#mask*,(ea)	-	-	-	-	-		-	-	-	8	-	8	12	14	12	16	-	-	-
							16				+4n		+4n	+4n	+4n	+4n	+4n			
							32				+8n		+8n	+8n	+8n	+8n	+8n			
movem	(ea),#mask*	-	-	-	-	-		-	-	-	12	12	-	16	18	16	20	16	18	-
							16				+4n	+4n		+4n	+4n	+4n	+4n	+4n	+4n	
							32				+8n	+8n		+8n	+8n	+8n	+8n	+8n	+8n	
movep	Dx,d(Ay)	-	-	-	-	-	16	16	-	-	-	-	-	-	-	-	-	-	-	-
							32	24	-	-	-	-	-	-	-	-	-	-	-	-
movep	d(Ax),Dy	-	-	-	-	-	16	16	-	-	-	-	-	-	-	-	-	-	-	-
							32	24	-	-	-	-	-	-	-	-	-	-	-	-
exg	Rx,Ry	-	-	-	-	-	32	-	6	6	-	-	-	-	-	-	-	-	-	-
lea	(ea),An	-	-	-	-	-	32	-	-	-	12	-	-	16	20	16	20	16	20	-
swap	Dn	-	×	×	0	0	32	4	-	-	-	-	-	-	-	-	-	-	-	-
link	An,#displ	-	-	-	-	-	16	16	-	-	-	-	-	-	-	-	-	-	-	-
unlk	An	-	-	-	-	-	32	12	-	-	-	-	-	-	-	-	-	-	-	-

* #mask: Registerliste

BCD-Befehle		Flags					Datenlänge	Ausführungszeit (Taktzyklen)												
		x	n	z	v	c		-	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rx)	xxxx.W	xxxxxx.L	d(PC)	d(PC,Rx)	#Op
abcd	Dy,Dx	-	?	×	?	×	8	6	-	-	-	-	-	-	-	-	-	-	-	-
abcd	-(Ax),-(Ay)	-	?	×	?	×	8	18	-	-	-	-	-	-	-	-	-	-	-	-
sbcd	Dy,Dx	-	?	×	?	×	8	6	-	-	-	-	-	-	-	-	-	-	-	-
sbcd	-(Ax),-(Ay)	-	?	×	?	×	8	18	-	-	-	-	-	-	-	-	-	-	-	-
nbcd	(ea)	-	?	×	?	×	8	-	6	-	12	12	14	16	18	16	20	-	-	-

Systemkontrolle	Flags x	n	z	v	c	Datenlänge	⋮	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rx)	xxxx.W	xxxxxx.L	d(PC)	d(PC,Rx)	#Op
move (ea),CCR	x	x	x	x	x	16	-	12	-	16	16	18	20	22	20	24	20	22	16
andi #Op,CCR	x	x	x	x	x	8	20	-	-	-	-	-	-	-	-	-	-	-	-
ori #Op,CCR	x	x	x	x	x	8	20	-	-	-	-	-	-	-	-	-	-	-	-
eori #Op,CCR	x	x	x	x	x	8	20	-	-	-	-	-	-	-	-	-	-	-	-
π) move (ea),SR	x	x	x	x	x	16	-	12	-	16	16	18	20	22	20	24	20	22	16
move SR,(ea)	-	-	-	-	-	16	-	6	-	12	12	14	16	18	16	20	-	-	-
π) andi #Op,SR	x	x	x	x	x	16	20	-	-	-	-	-	-	-	-	-	-	-	-
π) ori #Op,SR	x	x	x	x	x	16	20	-	-	-	-	-	-	-	-	-	-	-	-
π) eori #Op,SR	x	x	x	x	x	16	20	-	-	-	-	-	-	-	-	-	-	-	-
π) move USP,An	-	-	-	-	-	32	4	-	-	-	-	-	-	-	-	-	-	-	-
π) move An,USP	-	-	-	-	-	32	4	-	-	-	-	-	-	-	-	-	-	-	-
chk	-	x	?	?	?	16	-	10	-	14	14	16	18	20	18	22	18	20	14
π) rte	x	x	x	x	x			20	-	-	-	-	-	-	-	-	-	-	-
π) reset	-	-	-	-	-			132	-	-	-	-	-	-	-	-	-	-	-
trap #nr	-	-	-	-	-			38	-	-	-	-	-	-	-	-	-	-	-
trapv	-	-	-	-	-			4	-	-	-	-	-	-	-	-	-	-	-
π) stop #Op	x	x	x	x	x			4	-	-	-	-	-	-	-	-	-	-	-

Programmkontrolle	Flags x	n	z	v	c	Datenlänge	⋮	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rx)	xxxx.W	xxxxxx.L	d(PC)	d(PC,Rx)	#Op
nop	-	-	-	-	-		4	-	-	-	-	-	-	-	-	-	-	-	-
bra Ziel	-	-	-	-	-	8,16	10	-	-	-	-	-	-	-	-	-	-	-	-
jmp (ea)	-	-	-	-	-	32	-	-	-	8	-	-	10	14	10	12	10	14	-
bsr Ziel	-	-	-	-	-	8,16	18	-	-	-	-	-	-	-	-	-	-	-	-
jsr (ea)	-	-	-	-	-	32	-	-	-	16	-	-	18	22	18	20	18	22	-
rtr	x	x	x	x	x		20	-	-	-	-	-	-	-	-	-	-	-	-
rts	-	-	-	-	-		16	-	-	-	-	-	-	-	-	-	-	-	-
Scc*	-	-	-	-	-	wahr	-	4	-	12	12	14	16	18	16	20	-	-	-
						falsch	-	6	-	12	12	14	16	18	16	20	-	-	-
bcc*	-	-	-	-	-	8	10 gesprungen												
							8 nicht gesprungen												
						16	10 gesprungen												
							12 nicht gesprungen												
dbcc* Dn, Ziel	-	-	-	-	-	16	12 Bed. erfüllt, nicht gesprungen												
							10 Bed. nicht erfüllt, gesprungen												
							14 Bed. nicht erf., nicht gesprungen												

*cc: Condition-Code

Ausführungszeiten der 68000-Befehle

Arithmetik	Flags x	n	z	v	c	Datenlänge	!	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rx)	xxxx.W	xxxxxx.L	d(PC)	d(PC,Rx)	#Op
add (ea),Dn	c	×	×	×	×	8,16	-	4	4	8	8	10	12	14	12	16	12	14	8
						32	-	8	8	14	14	16	18	20	18	22	18	20	16
add Dn,(ea)	c	×	×	×	×	8,16	-	-	-	12	12	14	16	18	16	20	-	-	-
						32	-	-	-	20	20	22	24	26	24	28	-	-	-
adda (ea),Dn	-	-	-	-	-	16	-	8	8	12	12	14	16	18	16	20	16	18	12
						32	-	8	8	14	14	16	18	20	18	22	18	20	16
addi #Op,(ea)	c	×	×	×	×	8,16	-	8	-	16	16	18	20	22	20	24	-	-	-
						32	-	16	-	28	28	30	32	34	32	36	-	-	-
addq #Op,(ea)	c	×	×	×	×	8,16	-	4	8	12	12	14	16	18	16	20	-	-	-
						32	-	8	8	20	20	22	24	26	24	28	-	-	-
addx Dx,Dy	c	×	×	×	×	8,16	4	-	-	-	-	-	-	-	-	-	-	-	-
						32	8	-	-	-	-	-	-	-	-	-	-	-	-
addx -(Ax),-(Ay)	c	×	×	×	×	8,16	18	-	-	-	-	-	-	-	-	-	-	-	-
						32	30	-	-	-	-	-	-	-	-	-	-	-	-
sub (ea),Dn	c	×	×	×	×	8,16	-	4	4	8	8	10	12	14	12	16	12	14	8
						32	-	8	8	14	14	16	18	20	18	22	18	20	16
sub Dn,(ea)	c	×	×	×	×	8,16	-	-	-	12	12	14	16	18	16	20	-	-	-
						32	-	-	-	20	20	22	24	26	24	28	-	-	-
suba (ea),An	-	-	-	-	-	16	-	8	8	12	12	14	16	18	16	20	16	18	12
						32	-	8	8	14	14	16	18	20	18	22	18	20	16
subi #Op,(ea)	c	×	×	×	×	8,16	-	8	-	16	16	18	20	22	20	24	-	-	-
						32	-	16	-	28	28	30	32	34	32	36	-	-	-
subq #Op,(ea)	c	×	×	×	×	8,16	-	4	8	12	12	14	16	18	16	20	-	-	-
						32	-	8	8	20	20	22	24	26	24	28	-	-	-
subx Dx,Dy	c	×	×	×	×	8,16	4	-	-	-	-	-	-	-	-	-	-	-	-
						32	8	-	-	-	-	-	-	-	-	-	-	-	-
subx -(Ax),-(Ay)	c	×	×	×	×	8,16	18	-	-	-	-	-	-	-	-	-	-	-	-
						32	30	-	-	-	-	-	-	-	-	-	-	-	-

Ausführungszeiten der 68000-Befehle

Arithmetik (Fortsetzung)	Flags x	n	z	v	c	Datenlänge	Ausführungszeit (Taktzyklen) i	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rx)	xxxx.W	xxxxxx.L	d(PC)	d(PC,Rx)	#Op
muls (ea),Dn	-	×	×	0	0	max.	-	70	-	74	74	76	78	80	78	82	78	80	74
mulu (ea),Dn	-	×	×	0	0	max.	-	70	-	74	74	76	78	80	78	82	78	80	74
divs (ea),Dn	-	×	×	×	0	max.	-	158	-	162	162	164	166	168	166	170	166	168	162
divu (ea),Dn	-	×	×	×	0	max.	-	140	-	144	144	146	148	150	148	152	148	150	144
clr (ea)	-	0	1	0	0	8,16	-	4	-	12	12	14	16	18	16	20	-	-	-
						32	-	6	-	20	20	22	24	26	24	28	-	-	-
neg/negx (ea)	c	×	×	×	×	8,16	-	4	-	12	12	14	16	18	16	20	-	-	-
						32	-	6	-	20	20	22	24	26	24	28	-	-	-
ext Dn	-	×	×	0	0	16,32	4	-	-	-	-	-	-	-	-	-	-	-	-
cmp (ea),Dn	-	×	×	×	×	8,16	-	4	4	8	8	10	12	14	12	16	12	14	8
						32	-	6	6	14	14	16	18	20	18	22	18	20	14
cmpa (ea),An	-	×	×	×	×	16	-	6	6	10	10	12	14	16	14	18	14	16	10
						32	-	6	6	14	14	16	18	20	18	22	18	20	14
cmpi #Op,Dn	-	×	×	×	×	8,16	-	8	-	12	12	14	16	18	16	20	-	-	-
						32	-	14	-	20	20	22	24	26	24	28	-	-	-
cmpm (Ax)+,(Ay)+	-	×	×	×	×	8,16	12	-	-	-	-	-	-	-	-	-	-	-	-
						32	20	-	-	-	-	-	-	-	-	-	-	-	-
tst (ea)	-	×	×	0	0	8,16	-	4	-	8	8	10	12	14	12	16	-	-	-
						32	-	4	-	14	14	16	18	20	18	22	-	-	-
tas (ea)	-	×	×	0	0	8	-	4	-	14	14	16	18	20	18	22	-	-	-

Ausführungszeiten der 68000-Befehle

Schiebe/Rot.-Bef.		Flags						Ausführungszeit (Taktzyklen)												
		x	n	z	v	c	Daten länge	i	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rx)	xxxx.W	xxxxxx.L	d(PC)	d(PC,Rx)	#Op
asl/asr	Dx,Dy	×	×	×	×	×		8,16: 6+2n 32: 8+2n												
asl/asr	#Op,Dy	×	×	×	×	×		8,16: 6+2n 32: 8+2n												
asl/asr	(ea)	×	×	×	×	×		-	-	-	12	12	14	16	18	16	20	-	-	-
lsl/lsr	Dx,Dy×	×	×	×	0	×		8,16: 6+2n 32: 8+2n												
lsl/lsr	#Op,Dy	×	×	×	0	×		8,16: 6+2n 32: 8+2n												
lsl/lsr	(ea)	×	×	×	0	×		-	-	-	12	12	14	16	18	16	20	-	-	-
rol/ror	Dx,Dy	-	×	×	0	×		8,16: 6+2n 32: 8+2n												
rol/ror	#Op,Dy	-	×	×	0	×		8,16: 6+2n 32: 8+2n												
rol/ror	(ea)	-	×	×	0	×		-	-	-	12	12	14	16	18	16	20	-	-	-
roxl/roxr	Dx,Dy	×	×	×	0	×		8,16: 6+2n 32: 8+2n												
roxl/roxr	#Op,Dy	×	×	×	0	×		8,16: 6+2n 32: 8+2n												
roxl/roxr	(ea)	×	×	×	0	×		-	-	-	12	12	14	16	18	16	20	-	-	-

Bitmanipulation		Flags						Ausführungszeit (Taktzyklen)												
		x	n	z	v	c	Daten länge	i	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rx)	xxxx.W	xxxxxx.L	d(PC)	d(PC,Rx)	#Op
bclr	Dn,(ea)	-	-	×	-	-	8	-	-	-	12	12	14	16	18	16	20	-	-	-
							32	-	10	-	-	-	-	-	-	-	-	-	-	-
bclr	#Op,(ea)	-	-	×	-	-	8	-	-	-	16	16	18	20	22	20	24	-	-	-
							32	-	14	-	-	-	-	-	-	-	-	-	-	-
bset	Dn,(ea)	-	-	×	-	-	8	-	-	-	12	12	14	16	18	16	20	-	-	-
							32	-	8	-	-	-	-	-	-	-	-	-	-	-
bset	#Op,(ea)	-	-	×	-	-	8	-	-	-	16	16	18	20	22	20	24	-	-	-
							32	-	12	-	-	-	-	-	-	-	-	-	-	-
bchg	Dn,(ea)	-	-	×	-	-	8	-	-	-	12	12	14	16	18	16	20	-	-	-
							32	-	8	-	-	-	-	-	-	-	-	-	-	-
bchg	#Op,(ea)	-	-	×	-	-	8	-	-	-	16	16	18	20	22	20	24	-	-	-
							32	-	12	-	-	-	-	-	-	-	-	-	-	-
btst	Dn,(ea)	-	-	×	-	-	8	-	-	-	8	8	10	12	14	12	16	12	14	8
							32	-	6	-	-	-	-	-	-	-	-	-	-	-
btst	#Op,(ea)	-	-	×	-	-	8	-	-	-	12	12	14	16	18	16	20	16	18	-
							32	-	10	-	-	-	-	-	-	-	-	-	-	-

Ausführungszeiten der 68000-Befehle

Logische Befehle		Flags						Ausführungszeit (Taktzyklen)												
		x	n	z	v	c	Datenlänge	-	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rx)	xxxx.W	xxxxxx.L	d(PC)	d(PC,Rx)	#Op
and	(ea),Dn	-	×	×	0	0	3, 16	-	4	-	8	8	10	12	14	12	16	12	14	8
							32	-	8	-	14	14	16	18	20	18	22	18	20	16
and	Dn,(ea)	-	×	×	0	0	8, 16	-	-	-	12	12	14	16	18	16	20	-	-	-
							32	-	-	-	20	20	22	24	26	24	28	-	-	-
andi	#Op,(ea)	-	×	×	0	0	8, 16	-	8	-	16	16	18	20	22	20	24	-	-	-
							32	-	16	-	28	28	30	32	34	32	36	-	-	-
or	(ea),Dn	-	×	×	0	0	8, 16	-	4	-	8	8	10	12	14	12	16	12	14	8
							32	-	8	-	14	14	16	18	20	18	22	08	20	16
or	Dn,(ea)	-	×	×	0	0	8, 16	-	-	-	12	12	14	16	18	16	20	-	-	-
							32	-	-	-	20	20	22	24	26	24	28	-	-	-
ori	#Op,(ea)	-	×	×	0	0	8, 16	-	8	-	16	16	18	20	22	20	24	-	-	-
							32	-	16	-	28	28	30	32	34	32	36	-	-	-
not	(ea)	-	×	×	0	0	8, 16	-	-	-	12	12	14	16	18	16	20	-	-	-
							32	-	-	-	20	20	22	24	26	24	28	-	-	-
eor	Dn,(ea)	-	×	×	0	0	8, 16	-	4	-	12	12	14	16	18	16	20	-	-	-
							32	-	8	-	20	20	22	24	26	24	28	-	-	-
eori	#Op,(ea)	-	×	×	0	0	8, 16	-	8	-	16	16	18	20	22	20	24	-	-	-
							32	-	16	-	28	28	30	32	34	32	36	-	-	-

Hinweise zu den Flags:

-	nicht verändert	1	immer 1
×	gesetzt	c	wie Carry-Flag
0	immer 0	?	undefiniert

sonst. Hinweise:

An, Ax, Ay .... Adressregister
Dn, Dx, Dy .... Datenregister
Rn, Rx, Ry .... Daten- oder Adressregister
PC .... Programmzähler
SR .... Statusregister
CCR .... Condition-Code-Register
SP .... Stackpointer (user/supervisor)
USP .... Userstackpointer
SSP .... Supervisorstackpointer
π) .... privilegierter Befehl

Anhang E

Ausführungszeiten der 68010/12-Befehle im Loop-Modus

		Loop fortgesetzt			Loop abgeschlossen					
					Bedingung erfüllt			Zähler abgelaufen		
Befehl	Op.-L.	(An)	-(An)	(An)+	(An)	-(An)	(An)+	(An)	-(An)	(An)+
clr	8.16	10	10	12	18	18	20	16	16	18
	32	14	14	16	22	22	24	20	20	22
nbcd	8.16	18	18	20	24	24	26	22	22	24
neg	8.16	16	16	18	22	22	24	20	20	22
	32	24	24	26	30	30	32	28	28	30
negx	8.16	16	16	18	22	22	24	20	20	22
	32	24	24	26	30	30	32	28	28	30
not	8.16	16	16	18	22	22	24	20	20	22
	32	24	24	26	30	30	32	28	28	30
tst	8.16	12	12	14	18	18	20	16	16	18
	32	18	18	20	24	24	26	20	20	22

		Loop fortgesetzt	Loop abgeschlossen	
			Bedingung erfüllt	Zähler abgelaufen
Befehl	Op.-L.	op (ea),(ea)*		
addx	8.16	22	28	26
	32	32	38	36
cmpm	8.16	14	20	18
	32	24	30	26
subx	8.16	22	28	26
	32	32	38	36
abcd	8	24	30	28
sbcd	8	24	30	28

* Quelle und Ziel (ea) ist (An)+ für cmpm und -(An) für alle anderen

Ausführungszeiten der Befehle des 68010 im Loop-Modus

		Loop fortgesetzt			Loop abgeschlossen					
					Bedingung erfüllt			Zähler abgelaufen		
Befehl	Op.-L.	(ea) ↓ An*	(ea) ↓ Dn	Dn ↓ (ea)	(ea) ↓ An*	(ea) ↓ Dn	Dn ↓ (ea)	(ea) ↓ An*	(ea) ↓ Dn	Dn ↓ (ea)
add	8,16	18	16	16	24	22	22	22	20	20
	32	22	22	24	28	28	30	26	26	26
and	8,16	-	16	16	-	22	22	-	20	20
	32	-	22	24	-	28	30	-	26	28
cmp	8,16	12	12	-	18	18	-	16	16	-
	32	18	18	-	24	24	-	20	20	-
eor	8,16	-	-	16	-	-	22	-	-	20
	32	-	-	24	-	-	30	-	-	28
or	8,16	-	16	16	-	22	22	-	20	20
	32	-	22	24	-	28	30	-	26	28
sub	8,16	18	16	16	24	22	22	22	20	20
	32	22	20	24	28	26	30	26	24	28

*nur Wort erlaubt, (ea) darf nur (An),-(An) oder (An)+ sein

		Loop fortgesetzt			Loop abgeschlossen					
					Bedingung erfüllt			Zähler abgelaufen		
Befehl	Op.-L.	(An)	-(An)	(An)+	(An)	-(An)	(An)+	(An)	-(An)	(An)+
clr	8,16	10	10	12	18	18	20	16	16	18
	32	14	14	16	22	22	24	20	20	22
nbcd	8,16	18	18	20	24	24	26	22	22	24
neg	8,16	16	16	18	22	22	24	20	20	22
	32	24	24	26	30	30	32	28	28	30
negx	8,16	16	16	18	22	22	24	20	20	22
	32	24	24	26	30	30	32	28	28	30
not	8,16	16	16	18	22	22	24	20	20	22
	32	24	24	26	30	30	32	28	28	30
tst	8,16	12	12	14	18	18	20	16	16	18
	32	18	18	20	24	24	26	20	20	22

Ausführungszeiten der Befehle des 68010 im Loop-Modus

		Loop fortgesetzt			Loop abgeschlossen					
					Bedingung erfüllt			Zähler abgelaufen		
Befehl	Op.-L.	(An)	-(An)	(An)+	(An)	-(An)	(An)+	(An)	-(An)	(An)+
s/r *	16	18	18	20	24	24	26	22	22	24

*asr,asl,lsr,lsl,ror,rol,roxr,roxl

Die move-Befehle im Loop-modus des 68010:

Byte- und Wortverarbeitung

move	Loop fortgesetzt			Loop abgeschlossen					
				Bedingung erfüllt			Zähler abgelaufen		
	Ziel								
Quelle	(An)	-(An)	(An)+	(An)	-(An)	(An)+	(An)	-(An)	(An)+
Dn	10	10	-	18	18	-	16	16	-
An*	10	10	-	18	18	-	16	16	-
(An)	14	14	16	20	20	22	18	18	20
(An)+	14	14	16	20	20	22	18	18	20
-(An)	16	16	18	22	22	24	20	20	22

* nur Wort erlaubt

Langwortverarbeitung:

move	Loop fortgesetzt			Loop abgeschlossen					
				Bedingung erfüllt			Zähler abgelaufen		
	Ziel								
Quelle	(An)	-(An)	(An)+	(An)	-(An)	(An)+	(An)	-(An)	(An)+
Dn	14	14	-	20	20	-	18	18	-
An	14	14	-	20	20	-	18	18	-
(An)	22	22	24	28	28	30	24	24	26
(An)+	22	22	24	28	28	30	24	24	26
-(An)	24	24	26	30	30	32	26	26	28

Sachwortverzeichnis

Notizen

Notizen

Notizen

Notizen

Notizen

Klein

68000 kompakt

Mit 10 Abbildungen

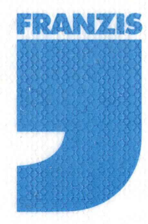
Umfang 256 Seiten

Dieses Buch erleichtert mit seinem klaren Konzept und mit seiner konsequenten Struktur das Auffinden von gesuchten Informationen über die 680xx-Prozessoren. Es beschränkt sich in der Darstellung auf die wesentlichen Dinge, die der Programmierer bei Assemblerarbeiten permanent braucht.

Aus dem Inhalt:

- eine klare Darstellung der Prozessor-Architektur
- eine umfassende Darstellung aller Adressierungsarten
- eine übersichtliche Darstellung aller Befehle
- eine präzise Angabe der Taktzyklen der einzelnen Befehle
- die Pinbelegungen aller gängigen Gehäusearten

Der Autor Rolf-Dieter Klein ist der Entwickler des inzwischen legendären NDR-Computers mit „Motorola-Herz". Seine Bücher und Fernsehsendungen sind die bekanntesten Anweisungen zum Selbstbau eines Computers. Von einem solchen Experten Kompaktinformationen zum 68000-Prozessor zu erhalten, garantiert Ihnen den hohen praktischen Nutzwert dieses Buches.





ISB N 3-7723-7632-0 DM +038.00